Tecniche di Debug Avanzate con PowerIntegrator

Indice

Tecniche di Debug Avanzate con PowerIntegrator

Scenario

Nel debug di applicazioni embedded odierne, ci si scontra con una complessità sempre crescente nei problemi da risolvere, al fine non solo di avere un software funzionante ed affidabile ma anche di ottenerne prestazioni soddisfacenti.

Nei design più sofisticati, l'interazione tra software ed hardware si fa molto complessa, e servono strumenti in grado di fornire una correlazione tra ciò che viene attuato dal software e la corrispondente risposta dell'hardware, un problema che nel debug "puramente" software non si presenta.

Un problema tipico in tali casi è controllare cosa transita sui bus di servizio tra le varie componenti dell'hardware a seguito dell'esecuzione di specifiche routine software.

In questi casi servirebbe uno strumento che permetta di analizzare lo stato dell'hardware nel tempo per poterlo correlare all'esecuzione del software, ovvero un **logic analyzer**.

Un numero sempre crescente di applicazioni inoltre fa un uso di diversi *protocolli di comunicazione* utilizzati sia per la comunicazione tra dispositivi locali (I2C, SPI, etc..) che per permettere lo scambio di dati con dispositivi remoti (seriali, bus industriali, etc…); anche in questo caso un logic analyzer può essere utilizzato per fare **protocol analisys**.

Sempre più di frequente inoltre, anche gli hardware progettati per applicazioni low-end o dalla complessità limitata, presentano microprocessori in grado di gestire dei kernel real-time, puntualmente utilizzati per l'enorme base di funzioni disponibili 'off-the-shelf' (ad esempio Connettività? -> Linux!), accorciando così i tempi di sviluppo, ma aggiungendo uno strato di complessità al debug delle applicazioni.

In questo genere di progetti, capita spesso che non sia disponibile una unità trace a bordo del silicio, ovvero i vincoli costruttivi del prodotto impediscono di sfruttarla anche se questa è presente; in questo caso servirebbe una unità che permetta di fare **bus trace** oppure di effettuare del **data logging** impiegando anche un numero limitato di risorse del chip, in questo caso un **logic analyzer** è ancora lo strumento ideale.

Le applicazioni in questo campo di un Logic Analyzer sono perciò molteplici e nello specifico, nel caso del debug di applicazioni embedded è un enorme vantaggio disporre di uno strumento **integrato** nell'ambiente di sviluppo software utilizzato, per l'intrinseco risparmio di tempo e impegno che ne deriva.

Scopo di questo documento

Lo scopo di questo documento è illustrare l'utilizzo di **PowerIntegrator** della linea PowerTools di Trace 32 come strumento di debug avanzato: un logic analyzer **integrato** nell'ambiente di debug **Lauterbach**.

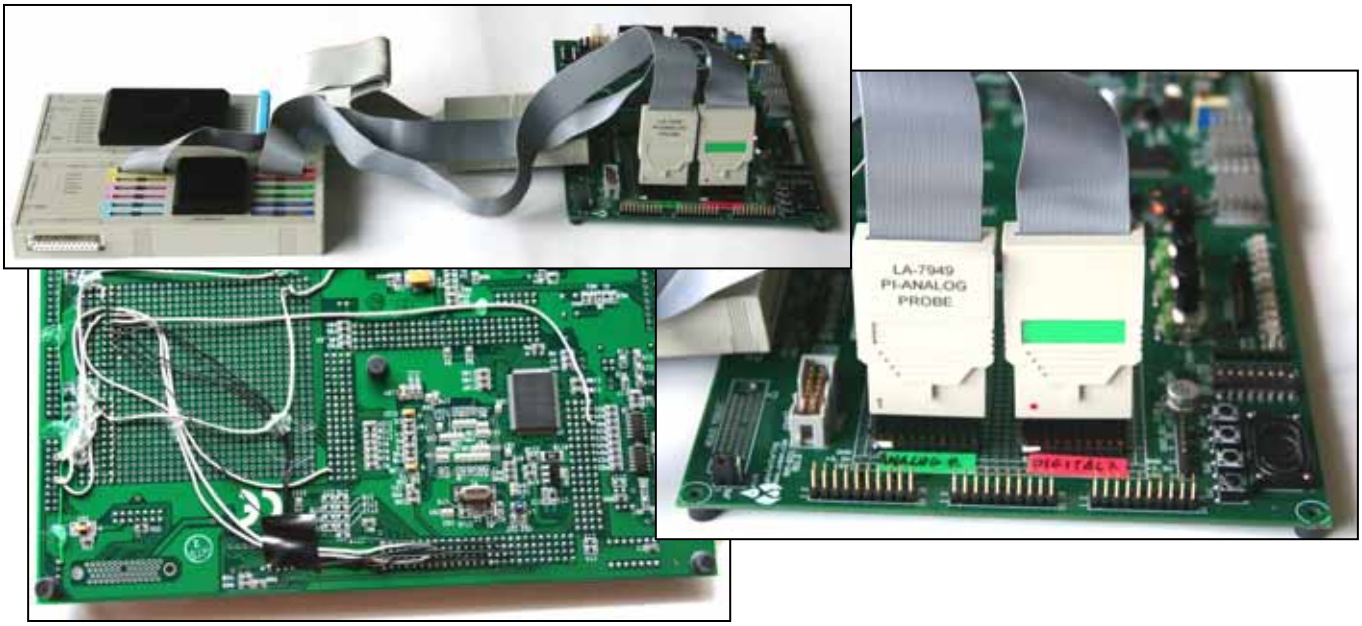
Il fine è trovare una soluzione ai problemi sopraccitati, illustrando attraverso alcuni esempi, corredati di spiegazioni e stralci di codice e script *PRACTICE,* le varie metodologie di impiego di PowerIntegrator.

I problemi illustrati in questa application note vogliono essere un esempio di situazioni realistiche nelle quali un logic analyzer come PowerIntegrator può tornare utile.

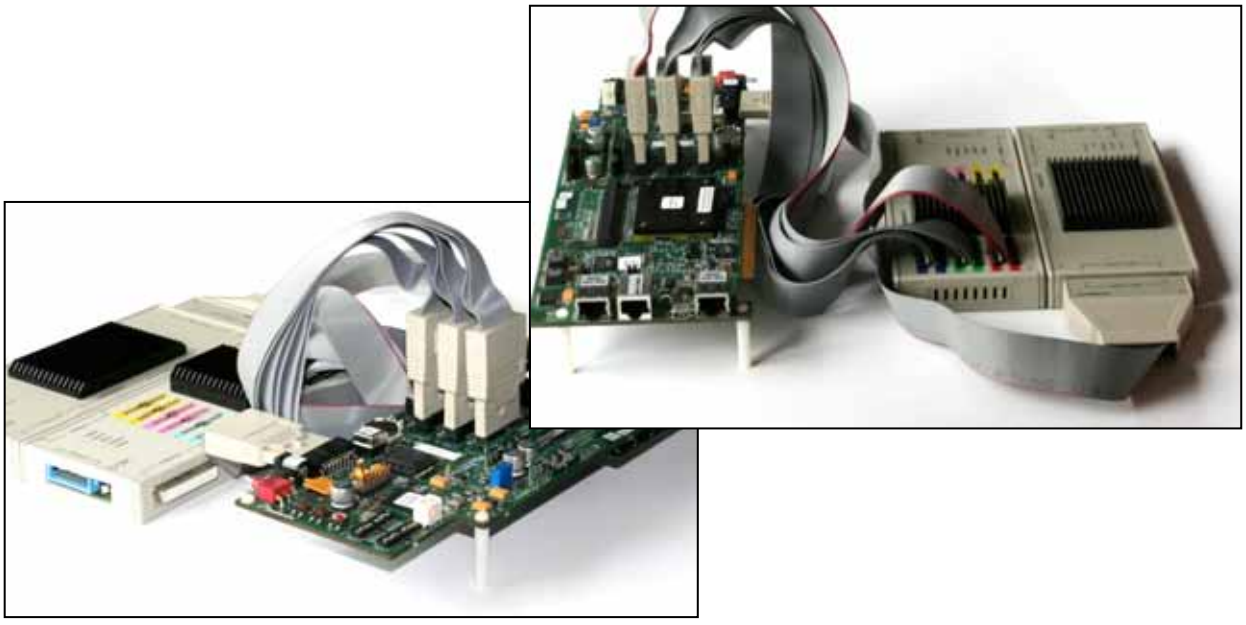
Hardware utilizzato

Gli esempi e le soluzioni esposte in questo documento sono state implementate su due hardware basati su architettura PowerPC, entrambi forniti da *Freescale Semiconductor*.

MPC5554EVB (http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC55xxEVB&fsrch=1),
Una board basata su MPC5554, cui sono stati aggiunti due connettori strip 10x2 di tipo berg a passo 0.127mm (100 mils) per la connessione dei probe del PowerIntegrator; nell'esempio abbiamo utilizzato un probe analogico ed uno digitale, per la misurazione di entrambi i tipi di grandezze.



MPC8360E-MDS (http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8360EMDS&fsrch=1),
Modular Development System per piattaforma PowerQUICC II Pro basata su di un MPC8360, dotata di 3 connettori MICTOR 38 pin per l'osservazione del bus di sistema (local bus) cui sono stati collegati i Mictor Probe Lauterbach disponibili per PowerIntegrator.



Interazione Hardware - Software

Correlazione tra Hardware ed eventi Software

L'utilizzo più immediato e semplice di un logic analyzer integrato nell'ambiente Trace32 è poter correlare in un diagramma temporale eventi hardware e software, compito solitamente estremamente difficile con uno strumento da tavolo, in cui la sincronizzazione all'esecuzione del software risulta generalmente approssimativa.

Con PowerIntegrator invece, tutto ciò è molto facile e praticamente immediato, semplicemente si fa uso della funzione **`Track`** di Trace32, che automaticamente sincronizza tutti i diagrammi basati sul tempo visibili sullo schermo, ne vediamo un esempio applicativo sulla MPC5554EVB.

In questo esempio un ADC a bordo del MPC5554 legge un valore di tensione determinato da un potenziometro, e genera un segnale PWM (Pulse Width Modulation) il cui *duty-cycle* è dipendente dal valore letto periodicamente dal convertitore.

Ciò che permette all'ambiente Trace32 di sincronizzare tutti i diagrammi basati sul tempo, è il fatto che ogni dispositivo controllato dall'ambiente T32 pone su ciascun campione acquisito una *timestamp*, ovvero una **etichetta temporale** che permette al debugger di determinare con precisione l'allineamento nel tempo di ogni record di ciascun dispositivo collegato.

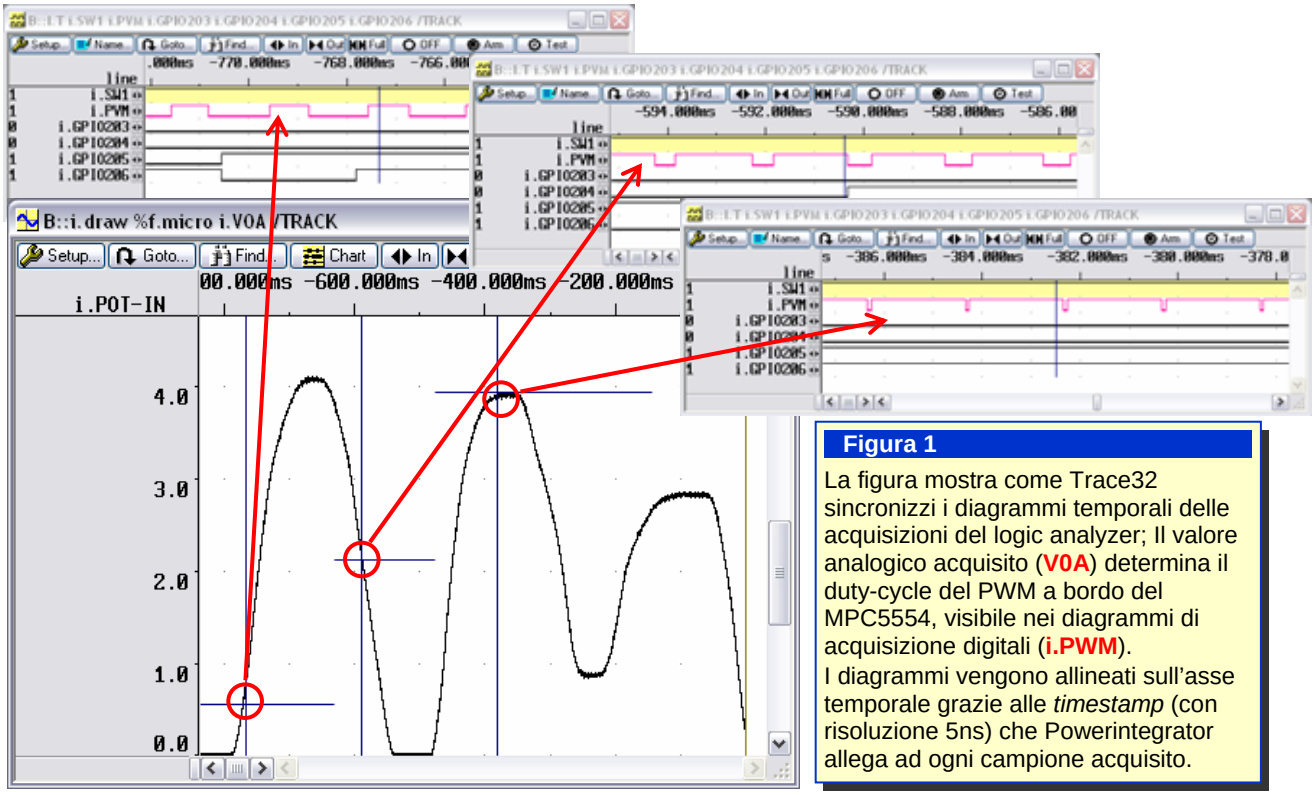


Figura 1

La figura mostra come Trace32 sincronizzi i diagrammi temporali delle acquisizioni del logic analyzer; Il valore analogico acquisito (**V0A**) determina il duty-cycle del PWM a bordo del MPC5554, visibile nei diagrammi di acquisizione digitali (**i.PWM**).
I diagrammi vengono allineati sull'asse temporale grazie alle *timestamp* (con risoluzione 5ns) che Powerintegrator allega ad ogni campione acquisito.

Essendo inoltre il processore MPC5554 dotato di una unità trace *Nexus* , abbiamo a disposizione il *real-time trace* del codice eseguito e grazie a questo meccanismo è immediato correlare eventi hardware con le specifiche routine software che li generano.

Ad esempio, nel semplice programma che segue (fig. 2.) un *Fixed Interval Timer* scatena un interrupt con contatore, ogni 100 ripetizioni un nibble esegue un conteggio da zero a sedici, che viene presentato su quattro pin GPIO: con il logic analyzer PowerIntegrator possiamo monitorare lo stato dei GPIO durante l'esecuzione, e successivamente analizzare il trace per capire quale routine ha causato il cambiamento.

Ora l'ambiente Trace32 deve correlare le informazioni provenienti da due dispositivi differenti, l'unità real-time trace ed il logic analyzer PowerIntegrator, e grazie al meccanismo delle timestamp già descritto i diagrammi temporali per entrambe le acquisizioni risultano sincronizzati con precisione.

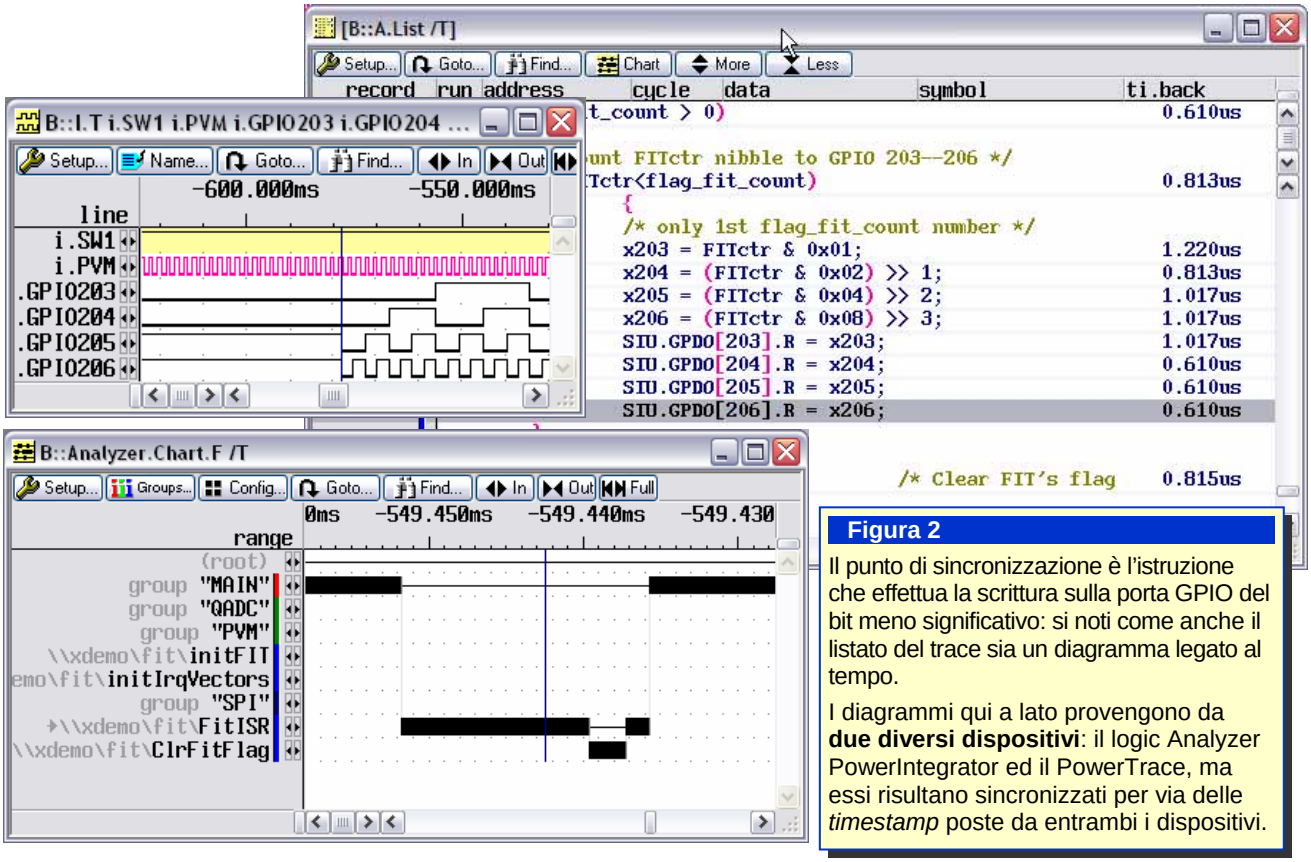


Figura 2

Il punto di sincronizzazione è l'istruzione che effettua la scrittura sulla porta GPIO del bit meno significativo: si noti come anche il listato del trace sia un diagramma legato al tempo.

I diagrammi qui a lato provengono da **due diversi dispositivi**: il logic Analyzer PowerIntegrator ed il PowerTrace, ma essi risultano sincronizzati per via delle *timestamp* poste da entrambi i dispositivi.

Naturalmente in questo semplice caso la sorgente del cambiamento è solo una e quindi il codice associato al conteggio è univoco, ma è facile pensare ad una situazione nella quale questo non sia vero, e può essere molto utile sapere "chi" ha fatto scatenare un determinato evento hardware tra tutti i possibili candidati nel software, per esempio quando ciò che è monitorato sia un evento *indesiderato*.

Trigger su eventi Hardware

Quanto visto in conclusione dell'esempio precedente, introduce un altro aspetto interessante nell'utilizzo di un oggetto come PowerIntegrator, che essendo **integrato** nell'ambiente Trace32 permette di monitorare eventi hardware ed eventualmente generare un **trigger** poi utilizzabile da Trace32 stesso (per esempio per fermare l'esecuzione del programma, o il trace).

PowerIntegrator dispone di una unità di trigger complessa programmabile per ottenere vari livelli di trigger e combinare gli effetti di segnali, timer e contatori al fine di far scattare l'impulso di trigger quando desiderato.

Prenderemo ad esempio un evento banale come la pressione di un pulsante sulla MPC5554EVB, il fronte di discesa del segnale su SW1 farà scattare un break nell'applicazione: le figure seguenti mostrano il setup del PowerIntegrator e del trigger di sistema.

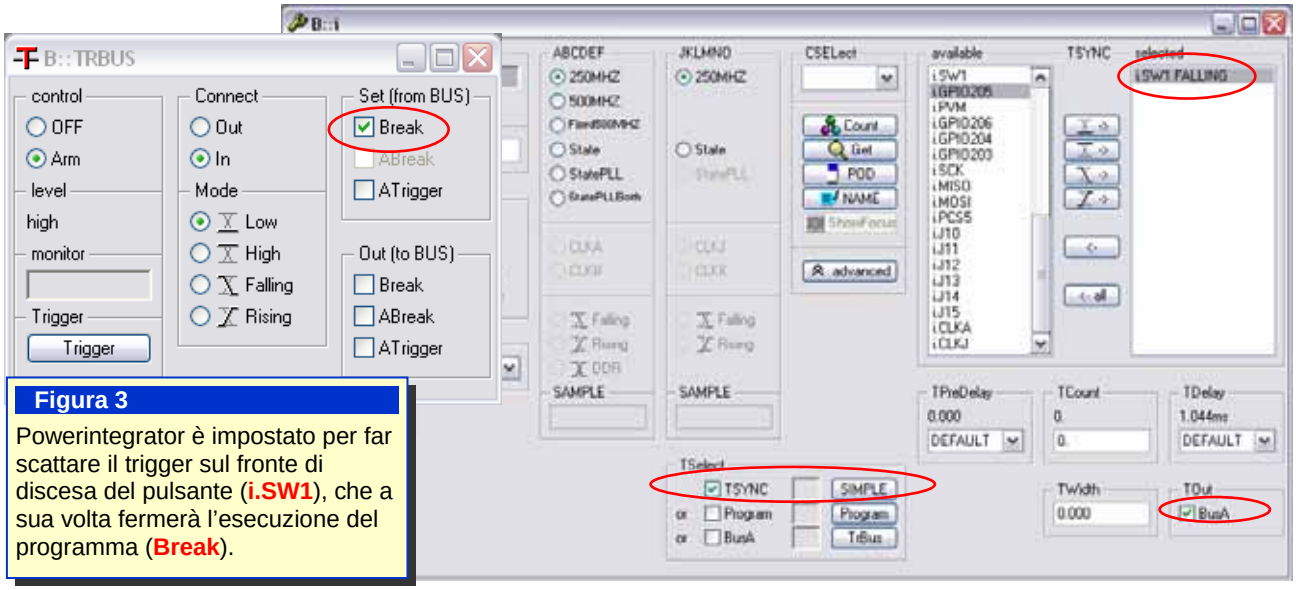


Figura 3

Powerintegrator è impostato per far scattare il trigger sul fronte di discesa del pulsante (**i.SW1**), che a sua volta fermerà l'esecuzione del programma (**Break**).

L'effetto visibile è che alla pressione del pulsante SW1 sulla board, il fronte di discesa generato sul canale monitorato (i.SW1) fermerà l'esecuzione del programma.

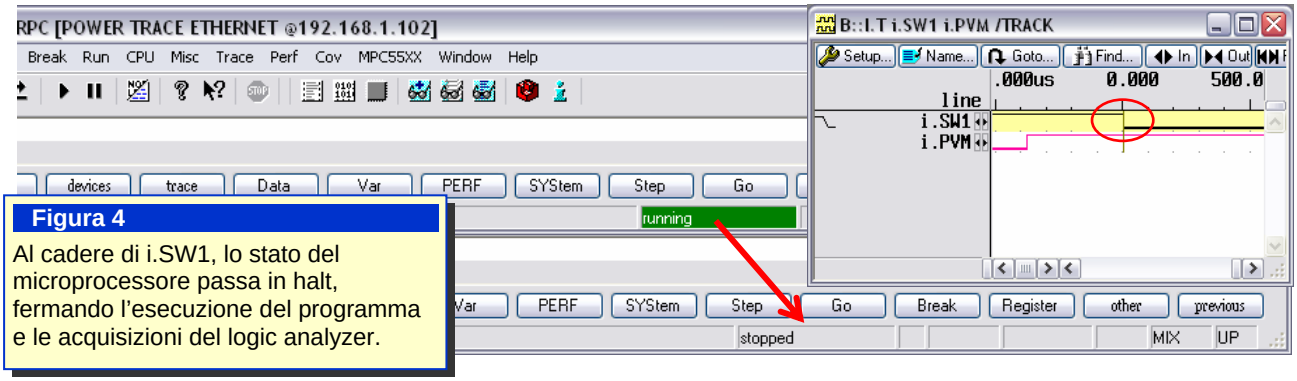


Figura 4

Al cadere di i.SW1, lo stato del microprocessore passa in halt, fermando l'esecuzione del programma e le acquisizioni del logic analyzer.

Trigger complessi

Basandosi sul medesimo concetto, è possibile creare trigger più sofisticati programmando la *complex trigger unit* del PowerIntegrator, per esempio facendo scattare il trigger solo in corrispondenza di una pressione del pulsante **di durata superiore a 100ms**; ciò è possibile grazie ai timer ed alla *state machine* disponibili nell'unità di trigger

Questo esempio riproduce una situazione in cui si vuole monitorare la durata di un evento e fermare l'esecuzione del programma nel caso quest'ultimo perduri oltre un determinato limite di tempo

Per fare ciò, le impostazioni del trigger di sistema rimangono le stesse ma si imposta il logic analyzer su `program` e si compila il seguente programma all'interno dell'unità di trigger:

```
SELECTOR switch i.SW1.Low      ; declaration section
TIMECOUNTER delay 100.ms       ; declare switch event and timer

L00:
counter.restart delay          ;Level 0: monitor if switch is
goto L01 if switch             ;pressed and change level

L01:
goto L00 if !switch            ;Level 1: go back if button
trigger.PODBUS if delay        ;is released before 100ms
                               ;otherwise issue trigger
```

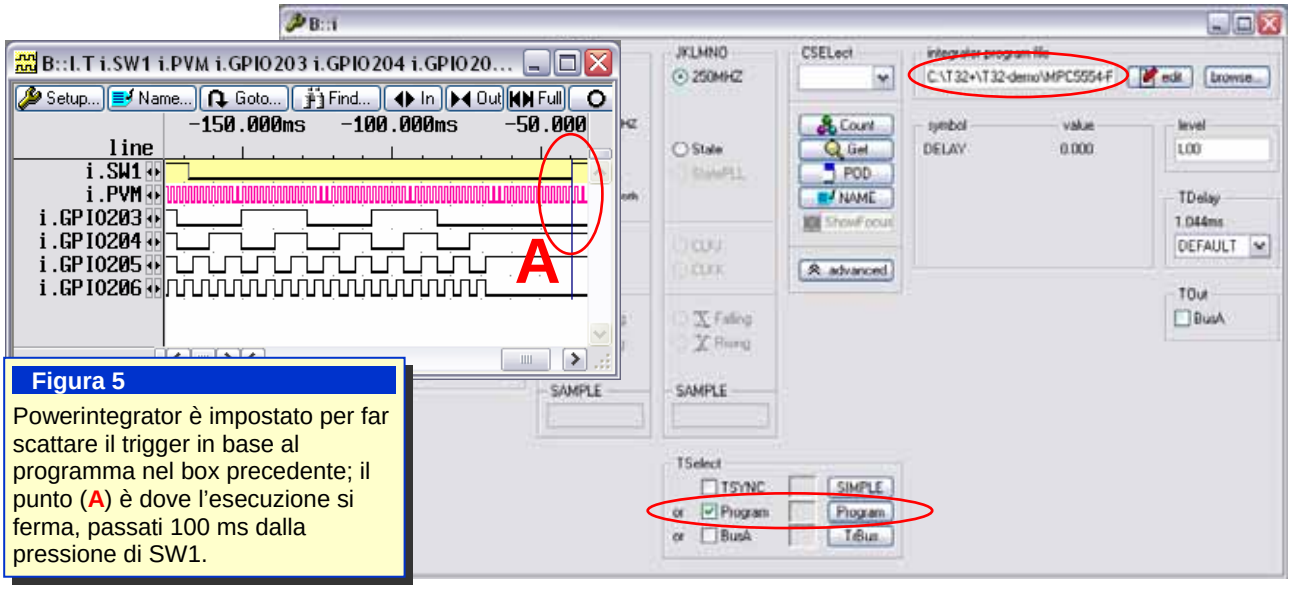


Figura 5

Powerintegrator è impostato per far scattare il trigger in base al programma nel box precedente; il punto (**A**) è dove l'esecuzione si ferma, passati 100 ms dalla pressione di SW1.

Partendo da questa base è facile comprendere come sia possibile costruire condizioni di trigger anche molto complesse con PowerIntegrator, permettendo di imporre condizioni di stop (ma non solo) legate al comportamento dello hardware, e delle corrispondenti attuazioni da parte del software.

Tecniche di Hardware Trace

L'applicazione più tipica di un logic analyzer è l'analisi di stati logici di gruppi di segnali; questa tecnica applicata campionando ogni ciclo del bus di sistema (*address* e *data*), permette la ricostruzione del codice eseguito dalla CPU, si ottiene cioè un *real-time trace* del programma con la tecnica del **Bus Trace**.

Allo stesso modo, si possono campionare solo specifici cicli del bus di sistema, o gruppi di segnali dedicati; in questo caso di parla di **Data Logging**, metodo con cui si ottengono informazioni meno dettagliate rispetto ad un trace "completo" ma necessita di un impiego minore di risorse.

Abbiamo realizzato degli esempi di questo utilizzo del PowerIntegrator sulla scheda MPC8360E-MDS di Freescale, con un processore PowerQUICC II pro, **sprovvisto** di una trace-port.

Questa evaluation board monta però tre connettori MICTOR™ per logic analyzer che consentono l'osservazione del bus locale, situazione ideale per l'utilizzo di PowerIntegrator.

Bus Trace

Osservando i cicli di bus eseguiti dal microprocessore, è possibile ricostruire il flusso del codice eseguito e i cicli di lettura e scrittura dati. In questo esempio abbiamo programmato nella memoria FLASH sul bus locale del MPC8360 della nostra board una semplice applicazione dimostrativa. Abbiamo poi collegato i probe del PowerIntegrator ai connettori per l'osservazione del bus locale, e lo abbiamo configurato per riconoscere i cicli di lettura effettuati campionando sul fronte di salita del segnale di *chip-select* della FLASH.

Abbiamo poi "istruito" il debugger a riconoscere I cicli campionati come cicli di ***fetch*** istruzione con il comando

```
i.disconfig.cycle "fetch" fetch address w.ADDR0-31 data w.DATA0-31
```

dove le word ADDR0-31 e DATA0-31 rappresentano il bus locale del MPC8360 (cfr. fig. 6)

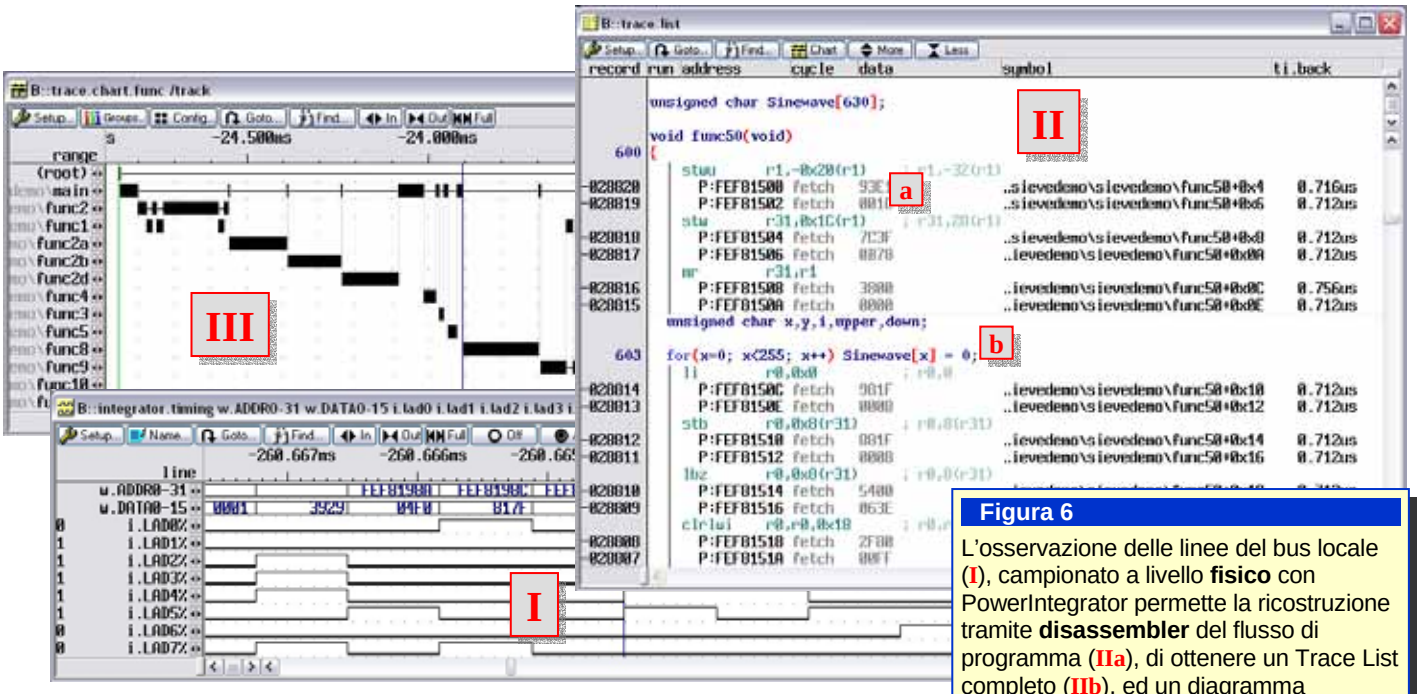


Figura 6

L'osservazione delle linee del bus locale (**I**), campionato a livello **fisico** con PowerIntegrator permette la ricostruzione tramite **disassembler** del flusso di programma (**IIa**), di ottenere un Trace List completo (**IIb**), ed un diagramma temporale ad alto livello (**III**).

La visibilità di tutti i cicli di esecuzione mette a disposizione dell'utente una mole di informazioni analizzabile con gli strumenti **standard** di Trace32: list, chart e statistiche (cfr. fig 6 e 7), che danno modo di analizzare dettagliatamente le performance del sistema.

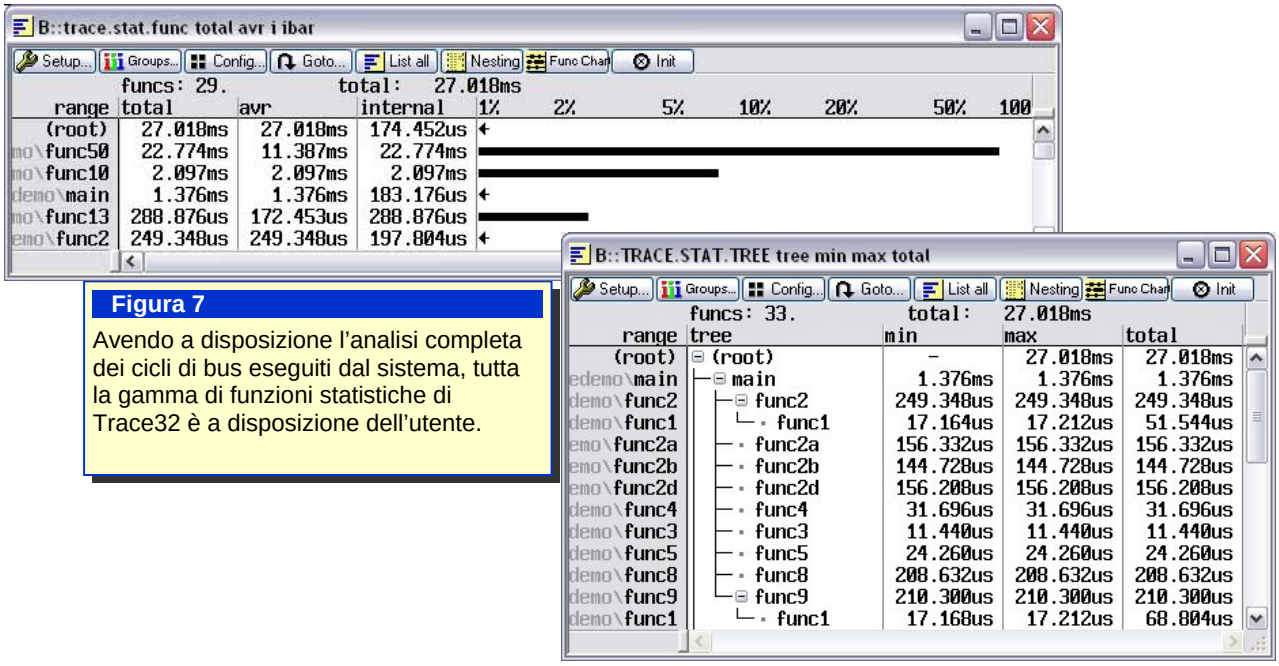


Figura 7

Avendo a disposizione l'analisi completa dei cicli di bus eseguiti dal sistema, tutta la gamma di funzioni statistiche di Trace32 è a disposizione dell'utente.

Flow Trace attraverso Branch-Trace messages

Una tecnica alternativa al Bus Trace è tracciare le sole informazioni di esecuzione **non lineare** del codice, ovvero ottenere informazioni sulle istruzioni *branch* colpite durante l'esecuzione: con un opportuno algoritmo si può poi ricostruire il *flusso* di programma, partendo dal presupposto che l'esecuzione del codice è sempre lineare tra un salto ed il successivo.

Il trace così ottenuto è completo e copre generalmente un lungo intervallo di tempo (in quanto meno campioni contengono più informazione) a discapito di una minore granularità dell'asse temporale, dato che si campionano record che rappresentano più cicli di bus.

Per fare un esempio concreto, implementiamo questa tecnica sulla board MPC8360E-MDS: nei core PowerPC è possibile generare un'eccezione ad ogni *asm-branch* colpito, ed introducendo quindi un'apposita *Interrupt Routine*, si trasmette a PowerIntegrator la locazione cui è avvenuta l'eccezione stessa, similmente a quanto fatto per il Bus Trace.

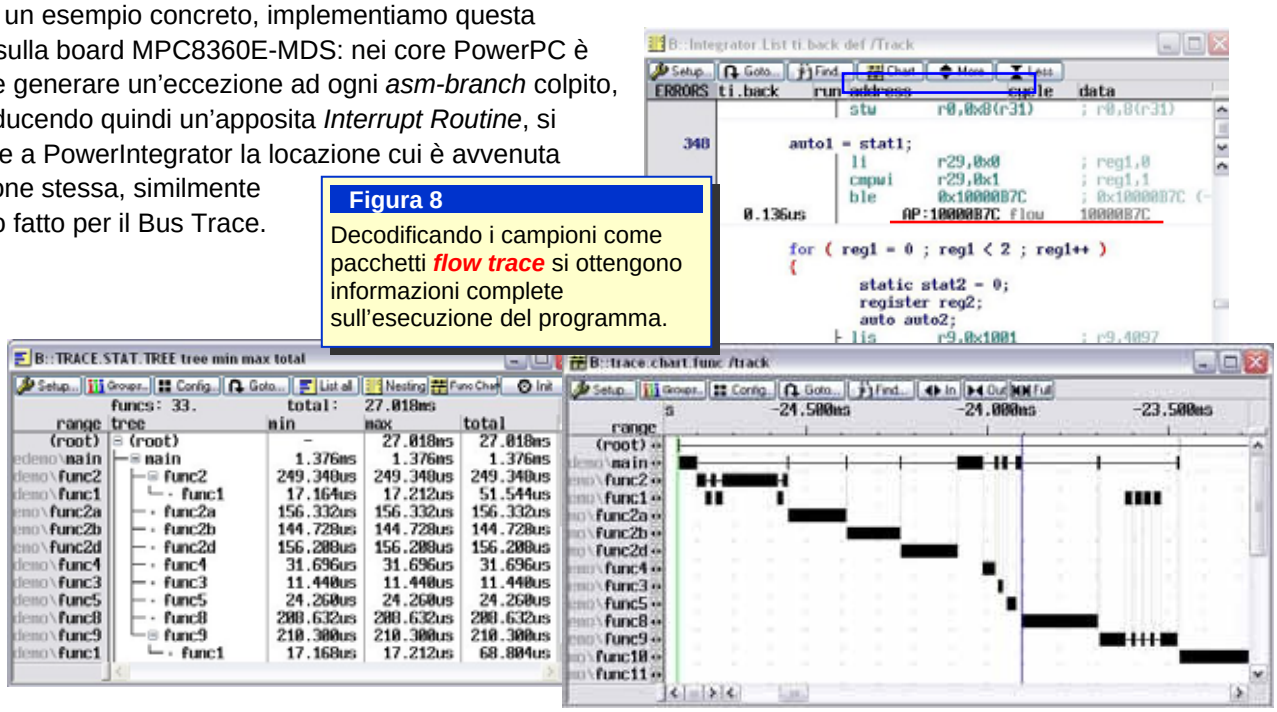


Figura 8
Decodificando i campioni come pacchetti ***flow trace*** si ottengono informazioni complete sull'esecuzione del programma.

La routine scrive nell'area definita dal *chip-select* della flash il valore del program counter che ha scatenato l'eccezione di *branch-trace*, che in un core PowerPC è contenuto nello *state recovery register* ***SRR0***.

"Istruendo" poi il debugger a riconoscere i cicli campionati come **flow**, possiamo ottenere il trace listing completo del programma (cfr. fig 8):

```
i.dc.cycle "flow" flow s i.nLWE0 low s i.nLWE1 low s i.LLA30 low a w.DATA0-31 d w.DATA0-31
```

Si noti come in questo caso l'**indirizzo** cui il dato viene "scritto" è ininfluente, è sufficiente che sia sempre quello, mentre l'**informazione** è contenuta nel solo **dato** (w.DATA0-31), che rappresenta il valore di SRR0 al momento in cui scatta la branch-trace exception.

Anche in questo caso, avendo informazioni complete sul codice eseguito, tutte le funzioni statistiche di Trace32 sono disponibili all'utente.

Data Logging

Come accennato in precedenza, un'altra tecnica spesso utilizzata è detta **data logging**, intesa come *cattura di una specifica informazione tramite la scrittura di un dato*; normalmente una tecnica di questo tipo può esser impiegata utilizzando un *buffer in memoria* nel quale **l'applicazione stessa** salva periodicamente i sopraccitati dati di interesse, per poi essere analizzati in un secondo tempo a target fermo.

Avendo però a disposizione un logic analyzer come PowerIntegrator, possiamo far sì che sia esso stesso a immagazzinare questi dati, **senza perciò "sprecare" risorse** del target, e solitamente con il vantaggio di una minore intrusione da parte del codice di logging.

Una situazione interessante si ha nel caso in cui sul target giri un **sistema operativo**, nel quale il logging può essere effettuato sia a livello del *kernel* che di applicazione (*user process*).

Per fare un esempio, utilizzeremo ancora la board MPC8360-MDS, sulla quale faremo eseguire un **Kernel Linux**, utilizzando la tecnica di data logging per diversi scopi:

- Con l'ausilio del *tick di sistema* cercheremo di fare una statistica qualitativa dell'impiego della CPU.
- Tracceremo tutti i momenti in cui Linux effettua un ***Task Switch***, e ne analizzeremo l'andamento

Al fine di ottenere questi risultati utilizzeremo del *codice di instrumentazione* all'interno del Kernel Linux, rappresentato dalle funzioni `T32_KloggerAddr` e `T32_KloggerData`, visibili in fig. 9.

La loro implementazione è estremamente semplice, e il principio di funzionamento è quello già illustrato nei paragrafi precedenti: entrambe faranno sì che la CPU effettui un ciclo di scrittura dei dati di nostro interesse nell'area definita dal *chip-select* della flash, facilmente campionabile con PowerIntegrator.

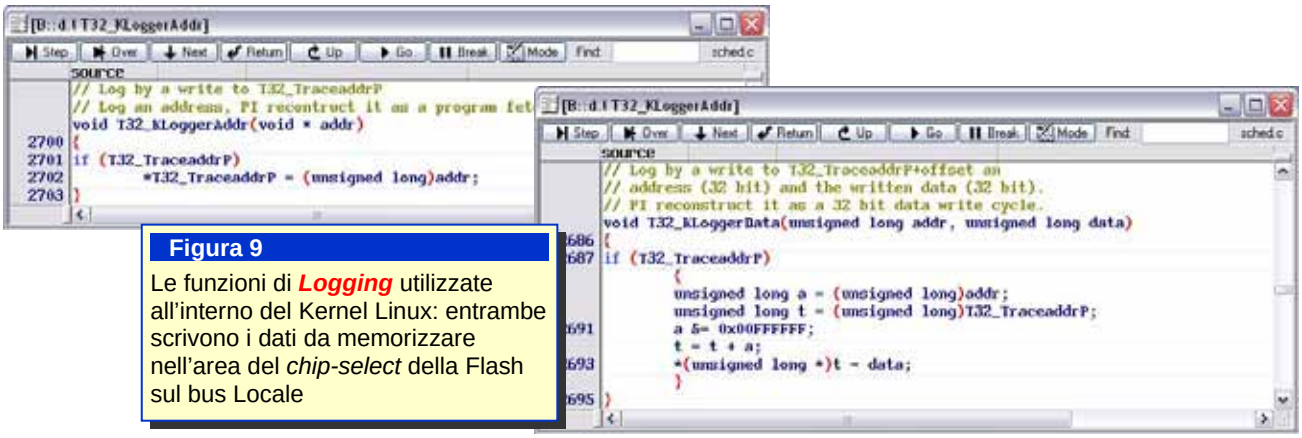


Figura 9
Le funzioni di ***Logging*** utilizzate all'interno del Kernel Linux: entrambe scrivono i dati da memorizzare nell'area del *chip-select* della Flash sul bus Locale

`T32_KLoggerAddr` esegue il logging di un **indirizzo**, allo scopo di tracciare il *passaggio per uno specifico punto del codice*, mentre `T32_KLoggerData` ha lo scopo di *tracciare* ***indirizzo e dato*** *di un ciclo di scrittura* eseguito dal microprocessore; chiamando in punti opportuni del codice queste due funzioni possiamo raggiungere gli scopi prefissati.

Come noto, Linux è un sistema operativo con scheduling dei processi in cui, allo scattare del timer di sistema (ogni 1ms nel nostro esempio), si realizza una divisione di tempo per i vari task in esecuzione; effettuando la chiamata a `T32_KLoggerAddr` in questo istante, implementiamo un sistema di logging a tempo che ci permette di avere una visione qualitativa delle performance assieme ad un trace indicativo, con un campionamento ogni 1ms (cfr. fig. 10)

Secondo lo stesso principio, instrumenteremo il codice affinché nello scheduler si tracci la *scrittura* del nuovo task corrente oltre che nelle strutture di controllo del kernel anche su PowerIntegrator, tramite `T32_KLoggerData`; per farlo, è sufficiente chiamare `T32_KLoggerData` nella funzione di scheduling, subito dopo Il passaggio

```
rp->curr = next
```

passando come parametri l'indirizzo del task magic (pointer) corrente ed il valore di `next` che vi viene scritto.

Grazie a queste informazioni siamo in grado di avere un trace qualitativo ed un preciso diagramma temporale dell'attività dei **task** (cfr fig. 10).

Affinché sia possibile inoltre sapere a quale spazio logico appartengano gli indirizzi presenti nel trace, facciamo sì che in corrispondenza dello scheduling di un nuovo processo venga trasmesso a PowerIntegrator anche lo *SpaceID* corrispondente, sempre con una `T32_KloggerData.`

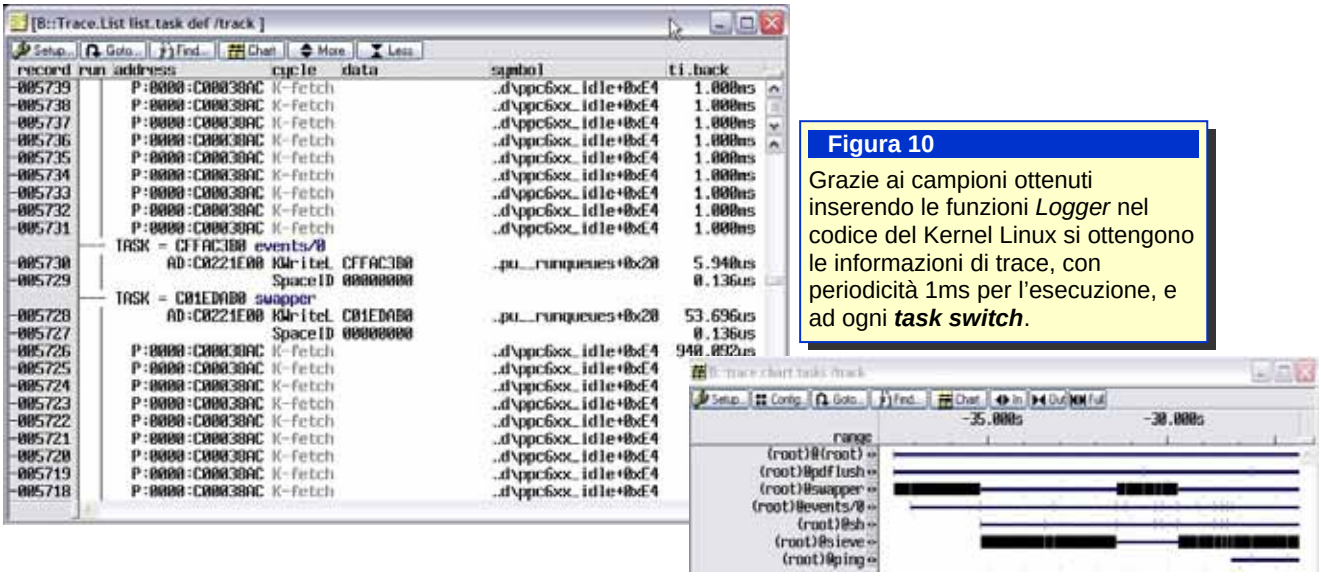


Figura 10

Grazie ai campioni ottenuti inserendo le funzioni *Logger* nel codice del Kernel Linux si ottengono le informazioni di trace, con periodicità 1ms per l'esecuzione, e ad ogni ***task switch***.

Naturalmente anche in questo caso istruiremo il disassembler, affinché PowerIntegrator riconosca i record campionati come cicli di bus del sistema; ecco un estratto dei comandi di configurazione:

```
;#### Kernel fetch logged at FF000000 #####
i.disconfig.cycle "K-fetch" fetch &sWriteL0 &KfetchQual a w.DATA0-31 &SIDsample spaceID
w.DATA0-31

;#### SpaceIDs logged at FF000008 #####
i.disconfig.cycle "SpaceID" write &sWriteL0 &KSID_Qual spaceID w.DATA0-31
```

Si può ottenere lo stesso risultato in *user-space*, ed è necessario in questo caso che l'applicazione possa scrivere nell'area della flash sul local bus; per fare ciò in Linux, l'applicazione deve poter scrivere ad indirizzi *fisici*, cosa solitamente non consentita, ma possibile (utilizzando il dispositivo `/dev/mem`).

Non è scopo di questa application note dettagliare il metodo utilizzato, basti sapere che il processo in questione è in grado di scrivere agli stessi indirizzi delle due funzioni già descritte, e Powerintegrator può catturare i campioni esattamente come per la tecnica adottata in *kernel-space*.

```
;#### User fetch logged at FF000004 #####
i.disconfig.cycle "U-fetch" fetch &sWriteL0 &UfetchQual a w.DATA0-31 &SIDsample spaceID
w.DATA0-31
```

Il risultato, è che ora si distinguono le operazioni fatte in *kernel-space*, *user-space* e relativi **dati**, permettendoci di utilizzare i consueti strumenti di analisi di Trace32. (vedi fig. 12)

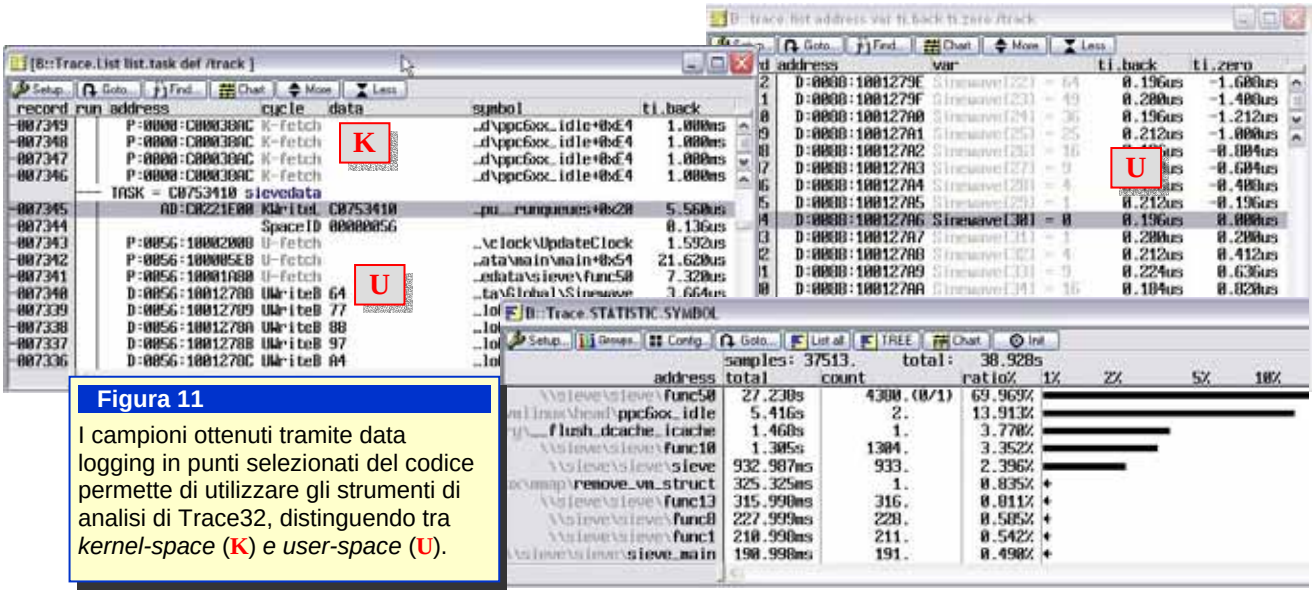


Figura 11

I campioni ottenuti tramite data logging in punti selezionati del codice permette di utilizzare gli strumenti di analisi di Trace32, distinguendo tra *kernel-space* (K) *e user-space* (U).

A seconda del tipo di applicazione quanto illustrato può essere sufficiente per estrarre tutte le informazioni desiderate sull'esecuzione del codice, sfruttando le sole capacità di ***disassembly*** di Trace32 e PowerIntegrator con la tecnica **Bus Trace.**

Una possibilità alternativa è fare affidamento alla tecnologia di **Protocol Analisys** di cui parleremo più avanti in questo documento.

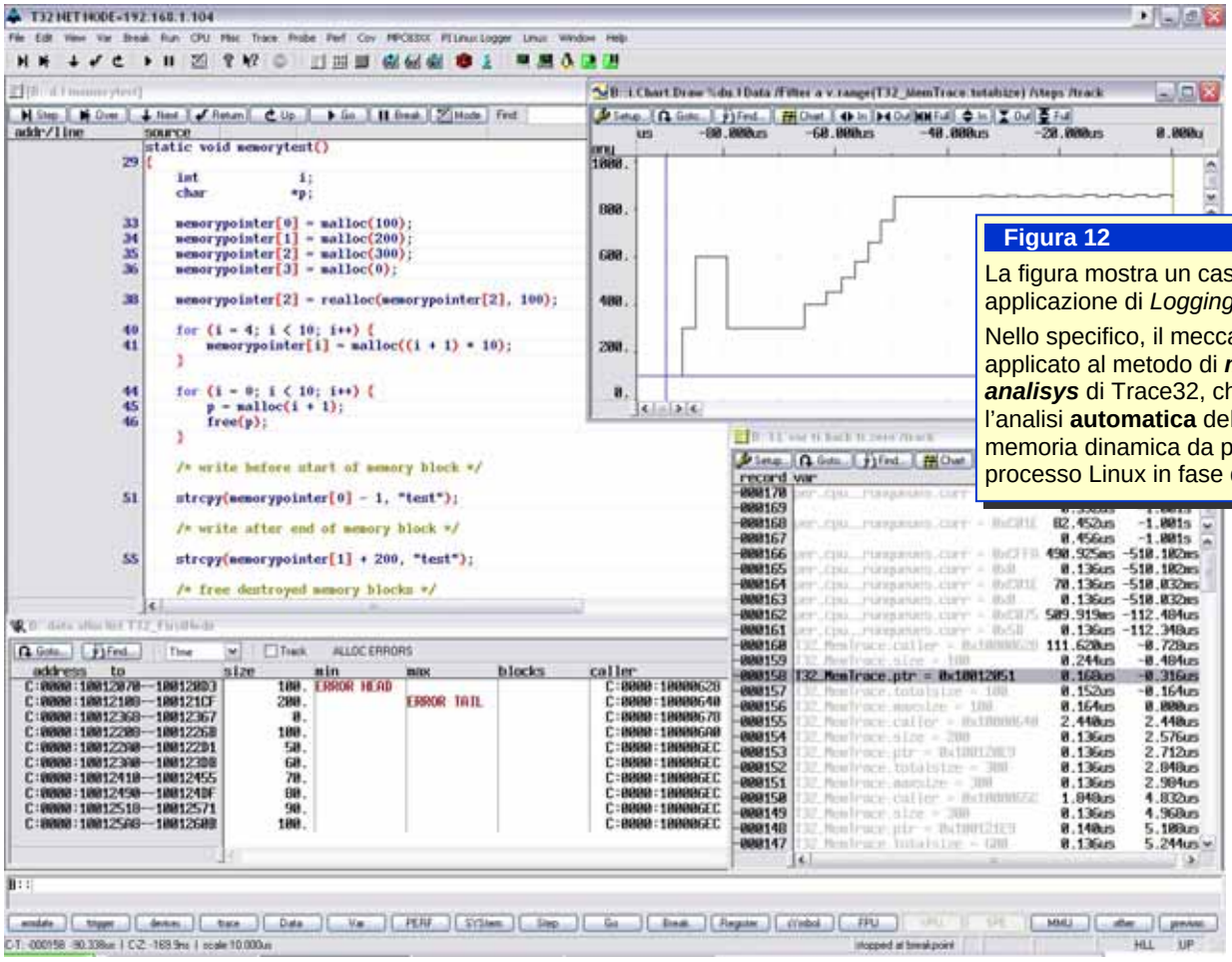


Figura 12

La figura mostra un caso realistico di applicazione di *Logging* in *user-space.*

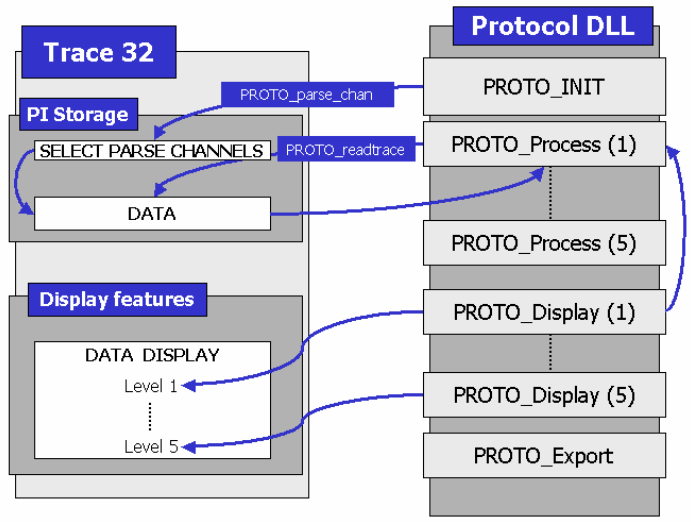
Nello specifico, il meccanismo è applicato al metodo di ***malloc analisys*** di Trace32, che permette l'analisi **automatica** dell'utilizzo della memoria dinamica da parte di un processo Linux in fase di debug.

Protocol Analysis

Potendo monitorare una grande quantità di segnali, è facile pensare al caso in cui questi ultimi facciano parte di un bus di trasmissione, un data bus od anche un qualsiasi protocollo proprietario di comunicazione, come capita spesso nelle applicazioni industriali.

All'interno di Trace32 la tecnologia ***protocol analyzer***, tramite una ***libreria dinamica*** (*DLL* o *shared object*) permette l'elaborazione dei dati acquisiti dal PowerIntegrator e la successiva visualizzazione in formato di protocollo (ad esempio di possono vedere i pacchetti transitanti sulle linee di un bus SPI).

Il sistema attualmente supporta protocolli quali *JTAG*, *I2C*, *Seriale* ed *SPI* ma può essere esteso per supportare **qualsiasi protocollo** dall'utente stesso, che può scrivere la una libreria **interprete** specifica per il suo protocollo proprietario, ed utilizzare le funzionalità di Trace32 per elaborarne i pacchetti; il principio sul quale si basa è l'utilizzo di un set di API per interfacciarsi all'ambiente Trace32, estrarre i dati dal trace storage di PowerIntegrator, elaborarli e visualizzarli tramite finestre predefinite.



La libreria deve essere programmata secondo poche regole ben precise: una funzione di inizializzazione (`PROTO_init`) predispone i canali che devono poi essere analizzati in lettura dalla funzione API `PROTO_readtrace` ed elaborati successivamente dalle funzioni di processo (`PROTO_Process`), in cui sono disponibili 5 differenti livelli di elaborazione. Infine, le funzioni di visualizzazione (`PROTO_display`) prendono i dati da quelle di processo e ne permettono il display in Trace32.

Oltre a ciò, possono essere definite anche delle funzioni di Export che permettono di salvare in un file i dati elaborati con il formato ritenuto più consono dall'utente.

Per esemplificare l'utilizzo di questa tecnologia, utilizziamo un programma che manda dei comandi sul bus SPI della MPC5554EVB al regolatore di tensione, accendendo e spegnendo la tensione di riferimento V1, osservandone l'andamento grazie al probe analogico; è utile notare come anche in questo caso la funzionalità di sincronizzazione di Trace32 allinei la visualizzazione dei pacchetti SPI con le acquisizioni analogiche del logic analyzer.

Anche in questo caso appare chiaro come l'utilizzo del PowerIntegrator sia fondamentale per ottenere una correlazione tra comportamento hardware ed esecuzione del software, ma appare anche evidente l'estrema **flessibilità** della tecnologia di protocol analysis, che è completamente **personalizzabile**.

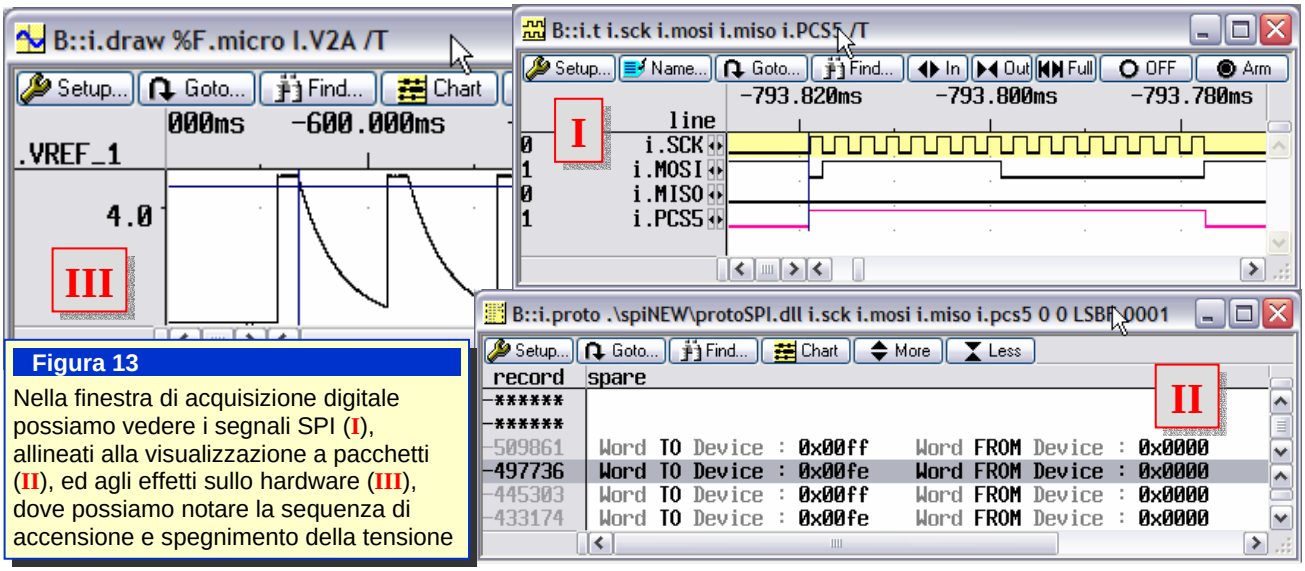


Figura 13

Nella finestra di acquisizione digitale possiamo vedere i segnali SPI (I), allineati alla visualizzazione a pacchetti (II), ed agli effetti sullo hardware (III), dove possiamo notare la sequenza di accensione e spegnimento della tensione

Data Logger via Protocol Analysis

Abbiamo visto come uno degli utilizzi più tipici di un logic analyzer come PowerIntegrator sia quello di tracciare l'esecuzione del software tramite la tecnica di **Bus Trace**, che permette la ricostruzione dei cicli di *fetch*, lettura e scrittura del microprocessore tramite l'osservazione diretta del bus di sistema; questa tecnica non è sempre applicabile ed anzi, nei target odierni generalmente non lo è, data l'invisibilità dei cicli di bus, oppure è soppiantata dai meccanismi di real-time trace sul silicio (es. cella ETM nei core ARM).

Spesso però molte utili informazioni sull'esecuzione del software si possono ottenere instrumentando il codice opportunamente, facendo cioè in modo che nei punti in cui desideriamo l'**osservabilità**, il codice esegua una chiamata ad una funzione opportuna (generalmente chiamata "Logger"), che renda disponibile l'informazione sul ciclo appena eseguito dal processore; essendo questa una tecnica **intrusiva** la frequenza con cui si chiama il logger deve esser opportunamente calibrata secondo le diverse esigenze, come già abbiamo avuto modo di illustrare in questa application note (cfr. Data Logging).

Se questa tecnica viene implementata facendo sì che questa informazione sia disponibile sullo **hardware**, si può pensare di utilizzare un protocollo specifico per far transitare le informazioni di esecuzione dal target verso il PowerIntegrator, ed utilizzare la protocol analysis per visualizzarle in Trace32, sfruttando le funzioni di analisi del *trace storage* disponibili, in opposizione a quanto fatto precedentemente tramite *disassembler*.

In questo programma di esempio, utilizzeremo una risorsa hardware come il bus SPI per far transitare informazioni sulla esecuzione del codice e dei cicli di scrittura e lettura di variabili.

Similmente a quanto fatto nel precedente programma, costruiremo una *protocol library* il cui secondo livello processerà i pacchetti SPI grezzi in ***Logger Data***, curandosi poi di riempire la seguente struttura dell'API di Trace 32:

```
typedef struct
{
    protoTime time;
    protoWord32 cycle;
    unsigned char datamask;
    unsigned char resv1;
    unsigned char flags;
    unsigned char resv2;
    protoWord64 address;
    protoWord64 data;
}
protoDiscycle,*protoDiscycleP;
```

La struttura **`protoDisCycle`** contiene le informazioni necessarie a ricostruire il singolo ciclo di bus del sistema, se Trace 32 trova informazioni valide in quest'ultima, le funzioni predefinite di listing, chart e di analisi statistica del trace storage risultano disponibili come se si avesse effettivamente a disposizione una unità di trace.

Nel nostro programma la routine in figura **`T32_SPI_Log`**, potrà essere chiamata ad ogni ciclo ritenuto necessario.

Ciò farà sì che sul PCS0 della MPC5554EVB transiteranno i pacchetti di tracciamento del programma.

Al livello 1 della nostra libreria, vedremo la codifica in pacchetti "grezzi" SPI, dei dati transitati, esattamente come nel precedente esempio di protocol analysis.

Ma ora questi pacchetti hanno significato anche ad un livello di astrazione più elevato: come si evince dal codice della routine, per ciascun ciclo sul bus SPI saranno trasmessi 3 pacchetti: un pacchetto **descrittivo** di sincronizzazione, uno contenente l'**indirizzo**, ed uno contenente il **dato**.

```
void T32_SPI_Log (char type, long address, long data )
{
  static vuint32_t spiCmdH,spiCmdL;

  if (t_spi_glob==0)
    return;

  /* send Synch packet */
  spiCmdL=(SPI_CMD_SYNCH)|(type<<1|SYNCH_ODD);
  DSPI_A.PUSHR.R = spiCmdL;
  while (DSPI_A.SR.B.TCF != 1){}

  /* send Address packet */
  spiCmdH=(SPI_CMD_ADDRH)|((address&0xFFFF0000)>>16);
  spiCmdL=(SPI_CMD_ADDRL)|((address&0x0000FFFF));
  DSPI_A.PUSHR.R = spiCmdH;
  DSPI_A.PUSHR.R = spiCmdL;
  while (DSPI_A.SR.B.TCF != 1){}

  /* send Data packet */
  if ((type==CYC_FETCH)||(type==CYC_READ_L)||(type==CYC_WRITE_L))
    {
    spiCmdH=(SPI_CMD_DATA_LH)|((data&0xFFFF0000)>>16);
    spiCmdL=(SPI_CMD_DATA_LL)|(data&0x0000FFFF);
    DSPI_A.PUSHR.R = spiCmdH;
    DSPI_A.PUSHR.R = spiCmdL;
    }
  else if ((type!=CYC_READ_B)&&(type!=CYC_WRITE_B))
    {
    spiCmdL=(SPI_CMD_DATA_W)|(data&0x0000FFFF);
    DSPI_A.PUSHR.R = spiCmdL;
    }
  else
    {
    spiCmdL=(SPI_CMD_DATA_B)|(data&0x0000FFFF);
    DSPI_A.PUSHR.R = spiCmdL;
    }
  while (DSPI_A.SR.B.TCF != 1){}
}
```

Il secondo livello della protocol ibrary, produrrà i dati all'interno della struttura **`protoDiscycle`**, e permetterà sia la visualizzazione dei cicli nella finestra di protocol analisys, sia nelle finestre relative alle funzioni predefinite di trace list e simili (come si può vedere in figura 14).

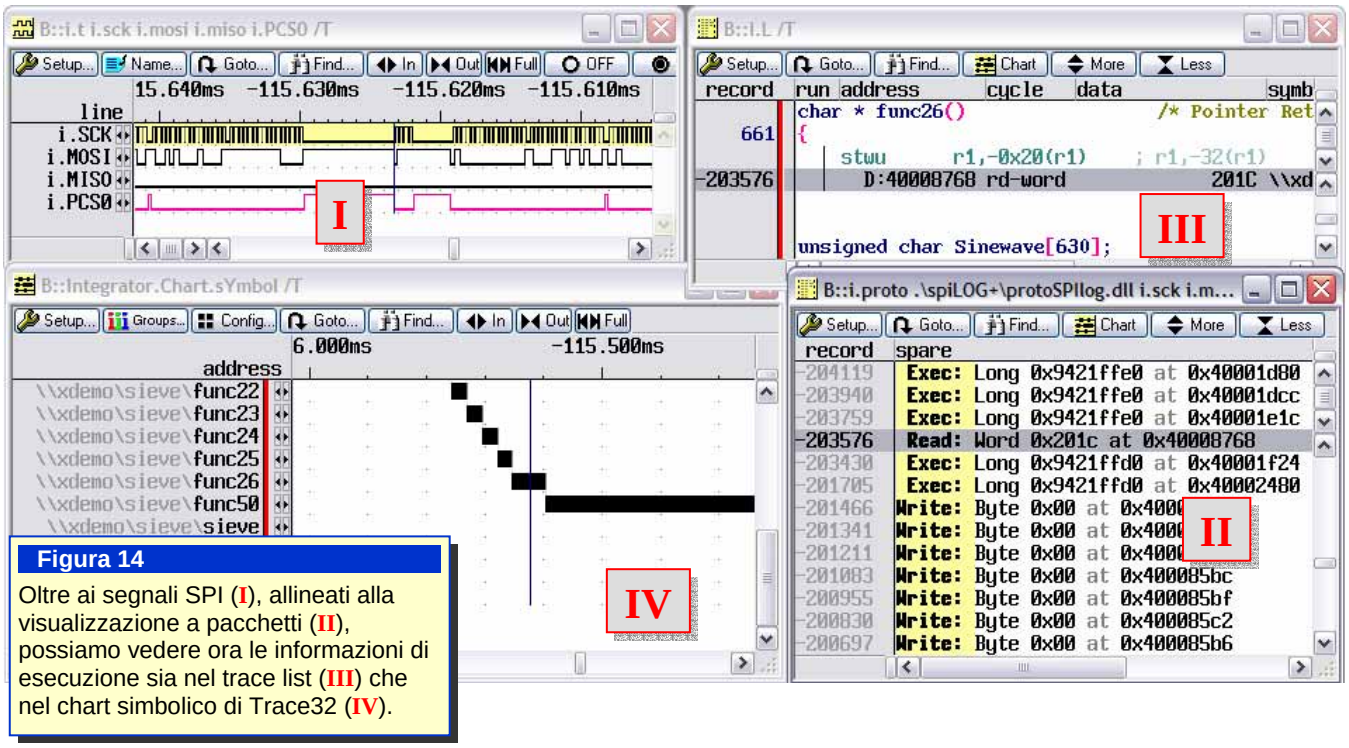


Figura 14

Oltre ai segnali SPI (I), allineati alla visualizzazione a pacchetti (II), possiamo vedere ora le informazioni di esecuzione sia nel trace list (III) che nel chart simbolico di Trace32 (IV).

Vale la pena confrontare quanto fatto in questo esempio di data logging via **Protocol Analisys** con la tecnica utilizzata parlando di data logging con **Bus trace**:

- appare evidente come in questo caso si utilizzino molte meno risorse hardware rispetto al bus trace, infatti servono solo **3 pin** dedicati del microprocessore, rispetto alla necessità di una **completa osservabilità del bus** di sistema nel caso del bus trace.
- L'**intrusione** nel codice è **maggiore** in quanto l'operazione di scrittura nell'area campionata non è più "atomica" (cioè un *singolo ciclo di scrittura della CPU*), ma ci si appoggia a risorse periferiche del microprocessore (lo SPI in questo caso).
  Esiste però la possibilità di **ottimizzare** anche questo aspetto, riducendo il più possibile il carico per il processore.
- La **profondità** di trace è inferiore poiché si utilizza un protocollo software, che comporta un **overhead** dipendente dalla complessità dello stesso, quindi servono **più campioni** per descrivere un singolo ciclo.
- Le prestazioni sono paragonabili, se come in questo caso si può utilizzare un protocollo sufficientemente veloce.

Le due tecniche hanno campi di applicazione simili e la scelta dipende in genere dalle **condizioni al contorno** con cui ci si deve confrontare, come spazio fisico ridotto che non permette di campionare l'intero bus, l'impossibilità di sfruttare alcun pin del microprocessore, la necessità di avere le informazioni con la minore intrusione possibile, ecc.

Conclusioni

In conclusione, dagli esempi mostrati possiamo capire come l'utilizzo di Powerintegrator metta a disposizione dell'utente un numero consistente di funzionalità molto utili ai fini del debug di applicazioni di ogni genere, sia per l'osservazione diretta dei fenomeni hardware in relazione all'esecuzione del codice, sia per la possibilità di essere programmato ed adattato all'analisi di ogni tipo di evento o protocollo.

Tutto ciò è possibile per la natura di questo logic analyzer, ovvero per il fatto di essere **integrato** nel sistema Trace32; in altre parole, grazie al fatto che **è possibile aggiungerlo per potenziare qualsiasi sistema Lauterbach oggi esistente**, sia esso un **PowerDebug** USB o un sistema completo **Powertrace II** di nuova generazione.

PowerIntegrator è lo strumento indispensabile che non può mancare in ogni laboratorio di sviluppo.

Powerintegrator – Caratteristiche Tecniche

- **Complex Trigger System**
- **Trigger I/O Synch con debugger**
- **Protocol Analyser:**
  CAN, USB, I2C, SPI, JTAG, Serial
  PCI, SDRAM + User Protocol Kit
- **Sampling Freq.** fixed 250MHz, fixed 500MHz, oppure da External clock
- **Trace Buffer** da 512K x 204 canali
  oppure 1024K x 102 canali
- Disassembler per **Bus trace**
- 3 Tipi di **Probe Digitali**: Mictor, Samtec, Berg
- **Probe Analogico**: 4 Voltage, 3 Current Input
- Probe per SDRAM, PCI, DDR, ESICON

Lauterbach Italia srl
Via E. Ferrieri 12
20153 Milano

Tel. +39 02 45490282
Fax +39 02 45490428
http://www.lauterbach.com

Autore: **Andrea Provasi**
support_it@lauterbach.it