Scuola universitaria professionale
della Svizzera italiana

Dipartimento
Tecnologie
Innovative

SSL

Parallax Propeller
Introduzione

Ivano Bonesana

Maggio 2007

(Doc. SSL-070503_DI-it)

Revisioni

Rev.	Data	Autore	Descrizione
0	03.05.2007	I. Bonesana	Prima stesura
0.1	15.05.2007	P. Ceppi, I. Bonesana	Correzioni e aggiunte
0.2	21.05.2007	P.Ceppi	Correzioni
0.3	21.05.2007	I. Bonesana	Aggiunta paragrafi programamzione, modificato esempi
1.0	18.6.2007	PCe	Rivisto abstract

Abstract

In recent years the reduction of feature size in silicon technology resulted in increased density of hardware systems. The research on microprocessor architectures has focused on devising and exploiting high level parallelism, including units replication and reutilization. Nowadays many processors are designed using a multi-core approach, i.e. integration of multiple processor cores on the same silicon chip.

Following this line of development, the Parallax Propeller integrates on the same chip 8 powerful 32-bit cores (the so called *Cogs*) which are able to run independetly and concurrently. Moreover the *Cogs* integrate a local memory and register structure, a VGA hardware unit and two 32-bits configurable timers.
The Propeller chip is a versatile product that may be used in a wide range of applications. In just a few hours, it is possible to develop a complete VGA-based application that would require much longer with other embedded platforms.

Although low-level software development for this particular hardware platform may sometimes appear difficult, most programming can be done at high level, using the language *Spin* proposed by Parallax.
The rich documentation is very useful and the Propeller forum as well as the freely available drivers and applications are an unsurpassed source of help. The Parallax-Propeller team is giving advice with very short turnaround time - great!

This is an overview of the architecture of the Propeller chip and its programming philosophy to set a starting point for future work at SUPSI-DTI. Examples are provided to show how to exploit the peculiarities of a multi-core chip. Focus is set on the programming language *Spin*.

Our overall assessment of the Propeller chip is good.

Acknowledgements

We want to thank Parallax Inc. for his assistance and for the donation of the Propeller Demo Board.

Ringraziamenti

Vogliamo ringraziare la Parallax inc. per l'assistenza fornitaci e per la donazione della Demo Board.

Indice

1 Introduzione

1.1 Il Propeller Chip

Il Propeller è un innovativo prodotto della Parallax caratterizzato da 8 cores a 32 bit integrati sul singolo chip. La struttura di base del componente è mostrata in figura 1. Ogni core[1] è un microprocessore completo che può funzionare indipendentemente dagli altri, grazie ad una propria memoria RAM che viene inizializzata via software (vedi sezione 2.2).
I componenti costruiti da più microprocessori in parallelo (cfr. architetture VLIW) non possono essere pogrammati ad alto livello (C, C++) dato che questi linguaggi non riescono a sfruttare appieno il parallelismo offerto dal sistema. Si tratta di un difetto strutturale dei linguaggi di programmazione concepiti per singolo processore che non offrono costrutti per parallelizzare le istruzioni a basso livello. Questi componenti sono programmati solitamente solo in assembly. Il Propeller non fa differenza: al linguaggio assembly viene affiancato un linguaggio dedicato chiamato *Spin*, vicino al *Basic* come sintassi, che permette di implementare algoritmi in modo relativamente semplice. Tra i costrutti del linguaggio *Spin* sono presenti alcune funzioni utilizzate per "istruire" i diversi Cog con il codice da eseguire.
Il Propeller dispone anche di risorse condivise tra i Cog mutualmente esclusive tra i Cog: memorie RAM e ROM. L'accesso a queste risorse è gestito da un meccanismo chiamato *Hub*: una logica round-robin che consente a ciascun Cog di usare a turno le risorse. Questo meccanismo limita l'indipendenza dei Cog e peggiora le prestazioni in termini di flusso di dati, perché più processori sono attivi e più tempo è richiesto per un accesso alle risorse condivise.
I pin di I/O sono organizzati in un'unica porta da 32 bit, sono condivisi tra i Cog e la loro direzione (input o output) dipende da come viene definita in ciascun Cog. Ai pin possono accedere contemporaneamente tutti i cog, una logica di arbitraggio evita possibili conflitti.
Questi temi saranno approfonditi nelle sezioni seguenti, facendo riferimento a [3].

1.2 L'assistenza Parallax

La Parallax offre un ottimo servizio di assistenza, per gli sviluppatori che utilizzano il Propeller, è aperto un forum sul sito ufficiale [1] molto frequentato. Il team di progettisti del chip ha fondato una comunità di sviluppatori molto attiva che risponde praticamente in tempo reale a qualsiasi dubbio o domanda. Ottima anche l'assistenza con esempi e documentazione.

[1] Nella terminologia Propeller si parla di *Cog* riferendosi ad un core.

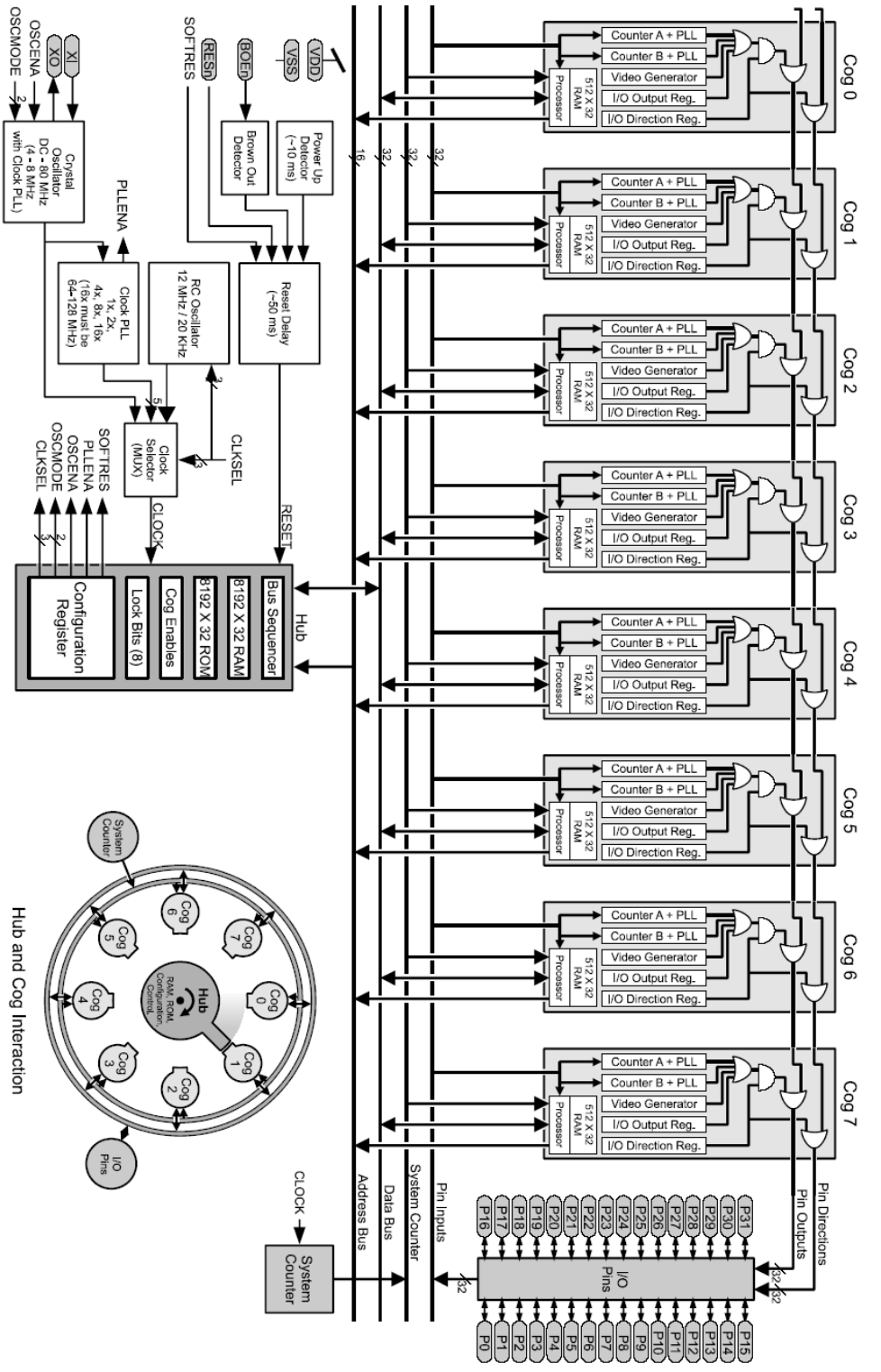


Figura 1: Schema del Propeller [2].

2 Descrizione tecnica

2.1 Il clock di sistema

Tre fonti sono disponibili per il clock di sistema del Propeller:

- un oscillatore inteno (RC): 20kHz o 12MHz;
- un'entrata per cristallo (XI - XO);
- il PLL basato su XI.

Le impostazioni del clock sono determinate dal registro CLK modificabile in software e a run-time[2]. Il timer di sistema è un contatore a 32 bit che incrementa il suo valore per ogni ciclo di clock. Sono diverse le tecniche software per il suo utilizzo e sono riportate nelle sezioni 3.1 e 3.3. Il valore del timer può essere letto contemporaneamente da tutti i Cog senza limitazioni accedendo al registro CNT. Il formato del registro CLK è riportato di seguito, per maggiori informazioni ci si riferisca a [2].

bit	7	6	5	4	3	2	1	0
name	RESET	PLLENA	OSCENA	OSCM1	OSCM0	CLKSEL2	CLKSEL1	CLKSEL0

RESET se '1' riavvia il chip azzerando il contatore;

PLLENA attiva ('1') o disattiva ('0') il PLL;

OSCENA attiva ('1') o disattiva ('0') il circuito del cristallo X1;

OSCMx(1:0) imposta la frequenza del quarzo;

CLKSELx(2:0) imposta la modalità del clock.

Tutti i parametri possono essere modificati in software mediante apposite istruzioni: per maggiori informazioni ci si riferisca al capitolo *CLK Register* del manuale [2].

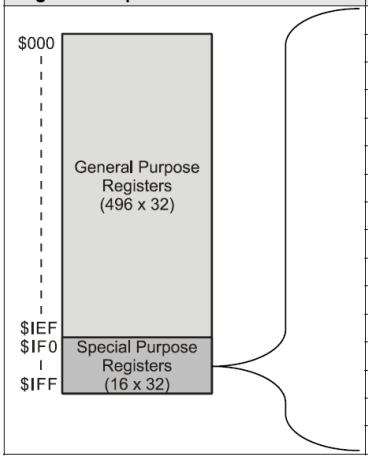
2.2 I *Cog*

Ogni *Cog* è un microcontrollore con un core a 32 bit le cui caratteristiche sono

- 2KB di memoria RAM con locazioni a 32 bit;
- due timer con relativi PLL per modificarne la frequenza di clock;
- una periferica per la generazione di segnali VGA;
- registri locali per I/O e per la configurazione.

L'attivazione e la disattivazione dei Cog può avvenire in runtime direttamente via software caricando il codice nella RAM locale che può essere utilizzata anche per variabili e dati. I registri interni dei diversi Cog sono mappati nella RAM nelle ultime 16 locazioni (vedi figura 2).
I PLL dei Cog possono variare la frequenza di lavoro fino ad un massimo di 80MHz con un quarzo a 5 MHz.

[2]La modifica del registro di controllo del contatore di sistema richiede circa 75μs



Cog RAM Map	Address	Name	Type	Description
	$1F0	PAR	Read-Only[1]	Boot Parameter
	$1F1	CNT	Read-Only[1]	System Counter
	$1F2	INA	Read-Only[1]	Input States for P31 - P0
	$1F3	INB	Read-Only[1]	Input States for P63- P32[2]
	$1F4	OUTA	Read/Write	Output States for P31 - P0
	$1F5	OUTB	Read/Write	Output States for P63 – P32[2]
	$1F6	DIRA	Read/Write	Direction States for P31 - P0
	$1F7	DIRB	Read/Write	Direction States for P63 - P32[2]
	$1F8	CTRA	Read/Write	Counter A Control
	$1F9	CTRB	Read/Write	Counter B Control
	$1FA	FRQA	Read/Write	Counter A Frequency
	$1FB	FRQB	Read/Write	Counter B Frequency
	$1FC	PHSA	Read/Write	Counter A Phase:
	$1FD	PHSB	Read/Write	Counter B Phase
	$1FE	VCFG	Read/Write	Video Configuration
	$1FF	VSCL	Read/Write	Video Scale

Note 1: Only readable as a Source Register (i.e. `MOV DEST, SOURCE`); read-modify-write not possible as a Destination Register.
Note 2: Reserved for future use.

Figura 2: Mappa della memoria RAM locale di ciascun Cog (Tab.6 [3]).

2.3 La memoria principale condivisa

La memoria principale è indirizzata in un unico spazio ed è divisa tra ROM e RAM, come mostra la figura 3. La ROM contiene le seguenti sezioni:

- Set di caratteri;
- Tavole:

  Logaritmo;

  Anti-Logaritmo;

  Seno.

- Bootloader e interprete Spin.

Il set di caratteri contiene 256 riquadri di 16x32 pixel utilizzati per visualizzare su VGA caratteri e simboli grafici corrispondenti ad un tipo di carattere denominato *Propeller Font Character Set* (v. appendice A).
Le tavole contengono valori numerici che consentono di ricostruire alcune funzioni. Questo permette, ad esempio, di produrre forme d'onda e segnali analogici, così come di effettuare calcoli che altrimenti richiederebbero l'uso di numeri a virgola mobile.
Il bootloader e interprete Spin è un firmware che viene utilizzato sia per attivare il codice programmato nella RAM del Propeller, sia ad eseguirlo. Lo Spin (v. sezione 3.1) è un linguaggio che deve venire interpretato a basso livello prima di poter essere eseguito. L'interprete svolge questo lavoro.
Il resto dello spazio di indirizzamento (8192 locazioni) è occupato dalla RAM condivisa tra i Cog in modo esclusivo: ciascun Cog può accedervi (lettura e scrittura) a turno. Il meccanismo che garantisce la sincronia di questo accesso è descritto nella sezione 2.4.

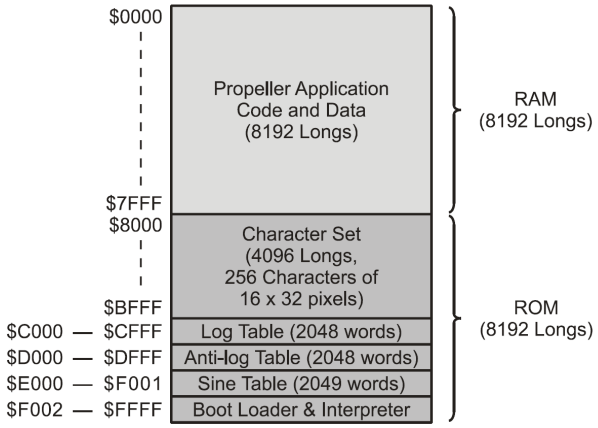


Figura 3: Mappa della memoria principale [3].

2.4 Hub

Nelle architetture con alto livello di parallelismo, solitamente, ad ogni core è associato un certo numero di canali per l'accesso alla memoria, in questo modo il sistema è in grado di ridurre la latenza d'accesso alla memoria condivisa rischiando però di compromettere i dati lì contenuti in quanto soggetti a possibili conflitti (WAW, RAW, ...).
Nel Propeller, l'accesso esclusivo alle risorse condivise è gestito da un meccanismo basato su di un principio di round-robin: ogni 16 cicli di clock un Cog (in sequenza da 0 a 7) ha accesso in lettura e scrittura alla memoria condivisa. La latenza di accesso alla memoria non è fissa, ma dipende dal numero di Cog attivi. Un'istruzione assembly che accede alla memoria condivisa richiede 7 cicli per essere eseguita. A questi va aggiunto un tempo di sincronizzazione, vale a dire che è necessario attendere che l'Hub inizi la "finestra di accesso"[3]. La sincronizzazione può durare fino a 15 cicli, per cui un accesso a memoria può richiedere:

- minimo 7 cicli di clock (caso migliore);
- massimo 22 cicli di clock (caso peggiore).

Da notare inoltre che l'Hub lavora ad un periodo di clock che è doppio rispetto al clock principale del chip.
Le figure 4 e 5 mostrano i due casi (rispettivamente il migliore ed il peggiore) per l'accesso a memoria con un'istruzione assembly che richiede 7 cicli per il completamento.

2.5 I pin di I/O

I pin possono essere usati da qualsiasi Cog in qualsiasi momento e contemporaneamente. La logica che consente di evitare la sovrapposizione di segnali sui pin di I/O è applicata combinando direttamente le richieste dei Cog, ognuno dei quali ha a disposizione un registro di 32 bit per stabilire la direzione della porta di I/O (*Direction*

[3]Hub Access Window: il tempo per il quale l'Hub è disponibile ad eseguire istruzioni di un dato Cog [3].

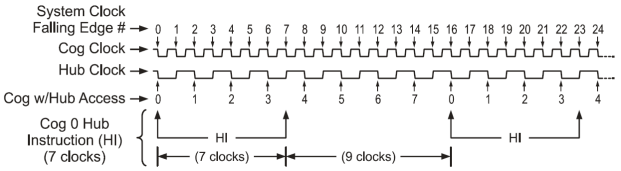


Figura 4: Accesso alla memoria condivisa, caso migliore [3].

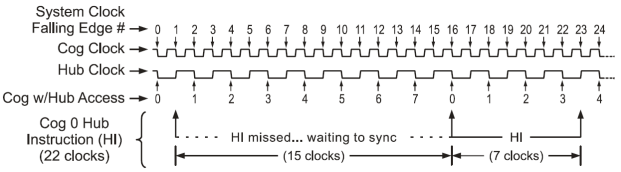


Figura 5: Accesso alla memoria condivisa: caso peggiore [3].

Register). Un valore logico '1' assegnato ad un bit del *Direction Register* imposta il corrispondente pin come *output*, un valore logico '0' lo imposta come *input*.
Sono valide le seguenti regole (v. circuito logico in figura 1):

- un pin impostato a '1' da uno o più Cog è visto come *output* (funzione logica OR);
- un pin è *output-low* (fornisce '0') se il risultato della funzione OR dei valori di output di ciascun Cog è '0' e il pin è utilizzato come output;
- un pin è *output-high* (fornisce '1') se il risultato della funzione OR dei valori di output di ciascun Cog è '1' e il pin è utilizzato come output.

I Cog dispongono anche di due registri locali, entrambi di 32 bit, adibiti uno ad input (`ina`) ed uno ad output (`outa`). Ad ogni lettura corrisponde lo stato attuale della porta di I/O. Ad ogni scrittura il valore viene inserito nel registro locale; la logica di arbitraggio propaga poi l'output sulla porta a seconda della direzione impostata dei pin nei differenti Cog.
Quando un Cog viene disattivato, i registri *Direction* e *Output* vengono azzerati, risultando quindi neutri nelle funzioni OR di cui sopra.

2.6 I timer

Ogni Cog dispone di due timer (*Counter*) configurabili a 32 bit, chiamati A e B. Lo schema a blocchi di un timer è riportato in figura 6 così come appare nell'*Application Note* [4]. Come verrà descritto in seguito, il Propeller non utilizza interrupt. I timer sono concepiti per utilizzare due pin come trigger (input) e/o come output.
Essenzialmente un timer è un contatore a 32 bit controllato da tre registri (CTRx, FRQx e PHSx) con a disposizione un moltiplicatore di frequenza PLL che permette di modificare la frequenza di funzionamento partendo da quella del Cog. Per informazioni più dettagliate ci si riferisca a [4].
Ad ogni ciclo una condizione d'incremento viene valutata e se ritenuta valida, il contatore incrementa il contenuto nel registro accumulatore PHSx, che conterrà il valore

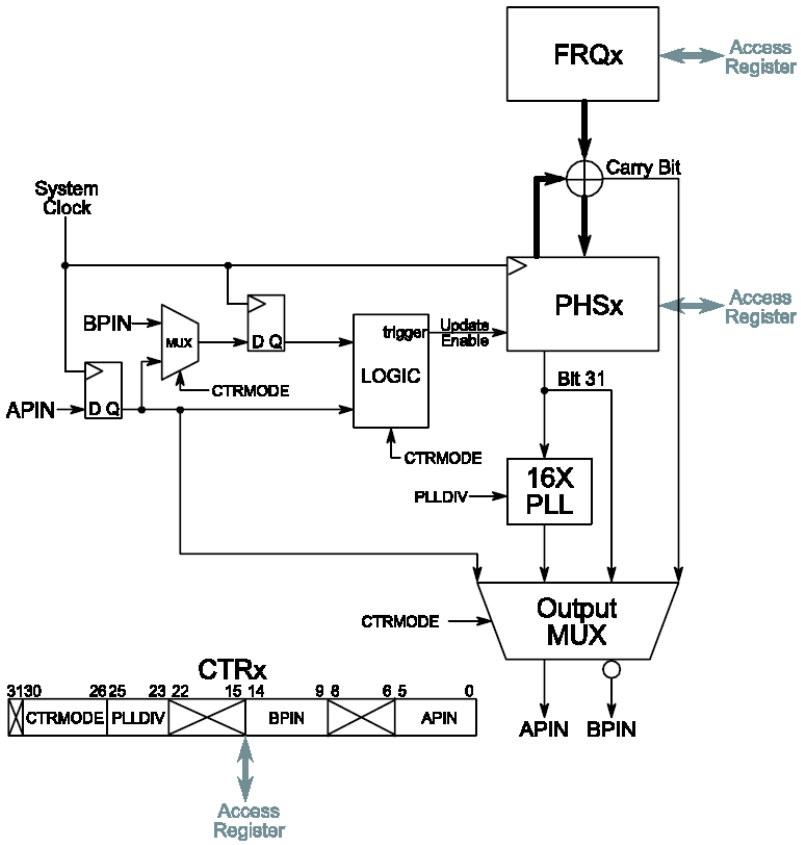


Figura 6: Schema a blocchi dei timer nei Cog [4].

corrente del timer che può essere letto o scritto da software. Il registro FRQx contiene invece il valore che viene aggiunto all'accumulatore ogni volta che la condizione d'incremento è valida. Questa condizione dipende dalla modalità operativa che viene impostata nel registro di controllo CTRx. Nello stesso CTRx si possono impostare il moltiplicatore del PLL e i pin di I/O.

31	30..26	25..23	22..15	14..9	8..6	5..0
-	CTRMODE	PLLDIV	-	BPIN	-	APIN

CTRMODE è la modalità di conteggio che include anche la condizione di comparazione (tab. 2 [4]);

PLLDIV è il valore del moltiplicatore del PLL (tab. 3 [4]);

APIN e BPIN sono i pin di input/output assegnati ai timer:

- input: possono fungere da trigger al posto del clock interno oppure per valutare lo stato di un segnale;
- output: funzionano press'a poco come un flag, quando la condizione di comparazione è valida cambiano valore.

L'impostazione del comportamento di un timer avviene modificando i valori dei relativi registri. Per manipolare i registri dei timer è da preferire l'uso di assembly con le istruzioni MOVI, MOVD, e MOVS (v. sezione 3.2.2). Questo a causa del tempo di interpretazione richiesto dalle istruzioni Spin che in questo contesto (timer 'cloccati' HW) possono dare risultati non prevedibili se si modificano i registri con istruzioni successive (ad es. prima si imposta il PLL poi la modalità, ...). In questo caso l'impostazione va eseguita con una singola istruzione. Per ulteritori dettagli si veda la sezione 3.1.3.
L'uso dei timer è riportato negli esempi della sezione 3.3 e in [5].

3 Programmazione

Il Propeller può essere programmato mediante un apposito tool di sviluppo, disponibile gratuitamente sul sito di Parallax Inc., che consente di lavorare sia in assembly che in *Spin*, il linguaggio proprietario che verrà descritto nella prossima sezione. Gli esempi e le applicazioni presentati in questa sezione sono stati implementati sulla *Propeller Demo Board* (figura 7) che offre un certo numero di periferiche e di interfacce tra cui: microfono, amplificatori audio, uscite TV e VGA, connettori per mouse e tastiera.

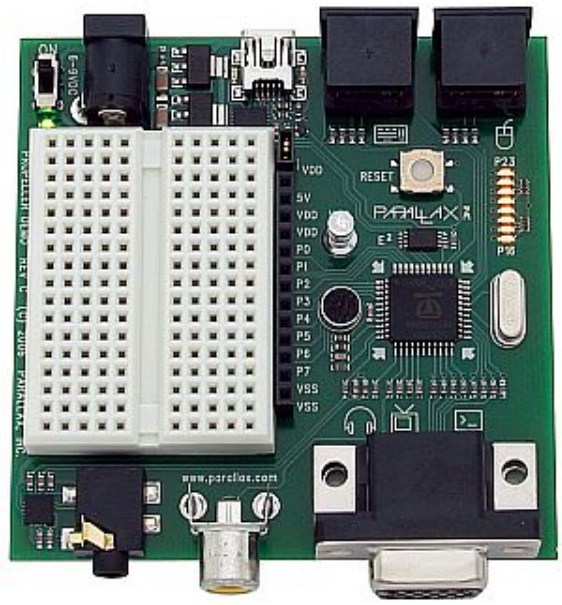


Figura 7: La *Propeller Demo Board*.

3.1 Il linguaggio *Spin*

Spin può essere definito come un "assembly ad oggetti". La sintassi richiama molto costrutti classici del C e del BASIC, mentre la struttura del programma è una fusione tra i linguaggi orientati agli oggetti e le tipiche sezioni che compongono un file eseguibile.
La base del linguaggio Spin è l'oggetto: un insieme di sezioni, che possono essere intese come i metodi (funzioni) dell'oggetto, alle quali si può accedere sia dall'esterno che dall'interno.
Le principali sezioni di un oggetto Spin sono:

CON che descrive le costanti;

VAR che comprende le variabili e i buffer di memoria condivisi tra tutte le funzioni dell'oggetto;

PUB che caratterizza una funzione ad accesso pubblico (anche dall'esterno dell'oggetto);

PRI che caratterizza una funzione ad accesso privato (usabile solo all'interno dell'oggetto);

DAT dove viene scritto codice assembly.

Esempi di come possa essere usato il linguaggio Spin, con una guida completa, sono disponibili in [2].
Un codice Spin non va direttamente a programmare l'hardware, viene bensì inviato all'interprete che risiede nella ROM del Propeller (vedi sezione 2.3) che lo converte in codice eseguibile a runtime. Per questo motivo nella programmazione con Spin si riscontra un ritardo (381 cicli di clock) nell'esecuzione delle istruzioni di un programma (v. sezione 3.1.2).

3.1.1 Attivare e disattivare un Cog

I Cog possono essere attivati da Spin caricando del codice nella loro memoria. L'attivazione avviene mediante la funzione `cognew`, la cui sintassi è la seguente:

```
cognew(funzione_Spin(parametri), StackPointer)
cognew(indirizzoASM, StackPointer)
```

Nella prima variante la RAM locale del primo Cog libero viene inizializzata con il metodo Spin indicato. Nella seconda variante il Cog viene inizializzato con il codice assembly preso dall'indirizzo indicato. Le funzioni restituiscono il numero (id) del Cog che è stato avviato.
Lo `StackPointer` è un indirizzo di memoria principale necessaria al Cog per eseguire il suo programma. In questa memoria si possono trovare dati condivisi tra i Cog. Le dimensioni dello `Stac` devono essere adeguatamente dimensionate per consentire al Cog di eseguire il suo programma. Lo `StackPointer` può essere usato in assembly per passare parametri al Cog. Un esempio tipico è il passaggio di indirizzi di memoria condivisa in modo da accedervi con le istruzioni `memread` e `memwrite` (v. sezione 3.2.5).
Per inizializzare uno specifico Cog si utilizza la funzione Spin `coginit`, analoga alla

precedente ma con indicato il numero del Cog da attivare. Per disattivare un Cog si utilizza la funzione `cogstop` che prende come argomento l'id del Cog da disattivare (il valore ritornato dalla `coginit`). La sintassi per usare le due funzioni è la seguente:

```
coginit(cogId, funzione_Spin(parametri), StackPointer)
cogstop(cogId)
```

L'uso dei parametri è descritto in sezione 3.2.5.

3.1.2 La temporizzazione

L'istruzione `WAITCNT` permette di sincronizzarsi con il clock di sistema del Propeller e mettere in attesa un Cog riducendo il consumo trattandosi di una sorta di modalità *low-power*. L'attesa è espressa in numero di cicli e quindi è necessario calcolare l'esatto valore corrispondente ad un certo intervallo di tempo. La sintassi Spin è la seguente (per assembly v. sezione 3.2.4):

```
waitcnt(N_CICLI)
```

In questo modo si istruisce il contatore di sistema ad attendere che il contatore di cicli di sistema (CNT) raggiunga il valore `N_CICLI` prima di riprendere l'esecuzione. Inserire un valore fisso di cicli da raggiungere non è però l'deale. Si preferisce in parecchi casi usare un'attesa differenziale, specificando per quanti cicli dal valore attuale del contatore di sistema (vedi sezione 2.1) con la seguente sintassi.

```
waitcnt(N_CICLI + CNT)
```

A causa del tempo necessario all'interprete Spin per eseguire l'istruzione, il valore di `N_CICLI` deve essere almeno 381, in caso contrario il sistema si blocca.
Spesso è necessario usare l'attesa all'interno di cicli che devono essere temporizzati. La tecnica tradizionale suggerisce di implementare quanto segue:

```
repeat
waitcnt(CNT + 50_000)
'codice da eseguire
```

Se si desidera un'attesa precisa, su [2] si fornisce il seguente esempio:

```
Time := CNT
repeat
waitcnt(Time += 50_000)
'codice da eseguire
```

Nel primo caso il tempo necessario per eseguire il codice contenuto nel ciclo 'ritarda' la successiva attesa. In questo modo l'esecuzione di ciascun ciclo risulta imprecisa. La variante, invece, aggiorna ad ogni ciclo il valore a cui il contatore di sistema deve arrivare. In questo modo il tempo di esecuzione del codice non influisce sull'attesa.
Esistono anche altri comandi per l'attesa (WAITPEQ, WAITPNE, WAITVID), per ulteriori dettagli si rimanda a [2].

3.1.3 Esecuzione e tempo di interpretazione

Un programam eseguibile per Propeller ha sempre una base (anche piccola) in Spin. La procedura di avvio di un Cog prevede che sia copiato nella RAM locale il codice dell'interprete che poi esegue il codice Spin del programma. L'esecuzione di Spin richiede l'interpretazione progressiva di ciascuna istruzione mentre per assembly l'esecuzione è direttamente affidata al processore.

3.2 Il linguaggio assembly

Più a basso livello, l'assembly del Propeller viene usato nella sezione DAT di un oggetto Spin e viene usato per programmare ogni Cog come un normale processore. In questo caso non c'è il ritardo nel tempo d'esecuzione dovuto all'interpretazione: per questa ragione si preferisce configurare alcune periferiche (come ad esempio i timer) in assembly anzichè in Spin.

3.2.1 Istruzioni condizionate

Come accade anche per altri microprocessori (v. ARM) il Propeller utilizza un assembly condizionale: ogni istruzione è subordinata ad una condizione che viene valutata prima dell'esecuzione. La struttura di un'istruzione generica si può rappresentare come:

```
<label> <condizione> <Istruzione> <flags>
```

Quando un'istruzione viene eseguita gli si può richiedere di affiggere uno dei bit classici del registro di stato (overflow, zero, ...) con i comandi `wx`[4]. I flag vengono poi valutati dalla successiva istruzione tramite la condizione `if_x` (v. [2]). Il seguente esempio mostra come possano essere utilizzate le istruzioni condizionate:

```
        test  t, sine_45  wc  'controlla se t = sin(45°)
if_c    neg   t, t            'se è così nega t
```

3.2.2 Configurazione di periferiche

Nel caso dei timer ci si trova confrontati ad un registro di 32 bit mappato in diversi gruppi di bit, alcuni dei quali non sono utilizzati (vedi sezione 2.6). Per facilitare l'inizializzazione sono state introdotte tre istruzioni particolari con funzioni simili all'istruzione MOV ma con alcune peculiarità:

MOVD (data field) assegna un valore ai bit 17..9 di un registro da 32 bit;

MOVI (instruction field) assegna un valore ai bit 31..23;

MOVS (source field) assegna un valore ai bit 8..0.

La sintassi è analoga per tutte e tre:

```
MOVx Destination, #Value
```

Dove `#Value` è una costante *literal* di 9 bit.

[4] Dove `x` rappresenta i flag `c` e `z`.

3.2.3 I *label* e le funzioni

Un *label* è un riferimento mnemonico associato ad un indirizzo di istruzione. Viene utilizzato per rendere più leggibile il codice.
Una funzione assembly è definita come un *label* associato ad un indirizzo di memoria dove inizia il codice da eseguire (*entry point*) e un *label* associato all'indirizzo dove il codice finisce. Il label di ritorno ha un formato particolare: `labelIniziale_ret`.
Seguendo questa particolare sintassi, l'assemblatore riesce a localizzare l'inizio e la fine delle routines associandone gli indirizzi. Un esempio di definizione di funzione è il seguente

```
miafunc      add a, b
             ...

miafunc_ret  ret
```

3.2.4 Temporizzazione in assembly

Analogamente a quanto discusso in sezione 3.1.2 per Spin, anche in assembly è possibile temporizzare l'esecuzione di un programma mediante l'istruzione `waitcnt`. La sintassi è la seguente: 3.2.4):

```
waitcnt target, N_CICLI
```

dove *target* è un registro che viene comparato con CNT, il counter di sistema. *N_ CICLI*, come nell'altro caso, è il numero di cicli che viene aggiunto al valore corrente del contatore di sistema.
L'istruzione mette in modalità di pausa il Cog che la utilizza fino a che il numero di cicli contenuti nel registro *target* non è stato raggiunto. A quel punto il contatore viene incrementato di *N_ CICLI* e il Cog riprende l'esecuzione normale.

3.2.5 Passaggio di parametri ai Cog

Quando si attiva un nuovo Cog è possibile specificare alcuni parametri da passare attraverso la funzione `cognew` e `coginit`, come mostrato in sezione 3.1.1. Questi parametri possono essere letti ed usati attraverso il registro `PAR`. Dato che non è possibile accedere direttamente alla memoria principale dall'interno dei Cog, i parametri sono usati in assembly come puntatori ad indirizzi esterni che possono essere usati dalle istruzioni `memread` e `memwrite`. L'esempio riportato su [2] è il seguente:

```
VAR
   long Shared                          'Shared variable (Spin & Asm)

PUB Main | Temp
   cognew(@Process, @Shared)            'Launch assy, pass Shared addr
   repeat
     <do something with Shared vars>

DAT
           org 0
Process    mov Mem, PAR                 'Retrieve shared memory addr
:loop      <do something>
           wrlong ValReg, Mem           'Move ValReg value to Shared
           jmp :loop
           jmp :loop

Mem        res 1
ValReg     res 1
```

In questo esempio il secondo parametro di `cognew` è utilizzato per passare l'indirizzo di memoria della variabile *Shared* che viene inserito nel registro `PAR`.

3.3 Esempi

3.3.1 Introduzione

In questa sezione verranno presentati alcuni brevi esempi esplicativi. Molti altri sono forniti da Parallax Inc. e dagli sviluppatori, sia sui forum sia su [5]. Inoltre, se si lavora in assembly è utile leggere [6] che riporta alcuni trucchi e accorgimenti da adottare per evitare errori. Molti sono anche gli oggetti disponibili per sfruttare alcune particolarità del Propeller: VGA e moduli per TV sono solo un esempio.
Al momento lo sviluppo sul chip è limitato dalla mancanza di un sistema di *debugging* efficiente e standardizzato. Ci sono alcuni progetti, anche open source, che si prefiggono di realizzare programmi in questo senso, ma Parallax Inc. non fornisce ancora una versione ufficiale. Durante lo sviluppo non si può quindi monitorare il comportamento del Propeller con un certo programma, in questo modo individuare eventuali errori di concetto può diventare alquanto difficile.
In questa sezione verranno esposti esempi sia in assembly che in Spin. L'utilizzo del linguaggio a più alto livello è senza dubbio più confortevole in quanto permette di utilizzare in modo pratico oggetti sviluppati da altri programmatori. In breve tempo si può infatti realizzare un'applicazione *multi-core* con visualizzazione su VGA. Tuttavia, quando si tratta di scendere a livello assembly per realizzare applicazioni particolari (acquisizione e trattamento di segnali, ...) si possono incontrare non poche difficoltà a causa dell'assenza di periferiche tipiche come DAC, ADC, UART, ... Gli esempi e le applicazioni proposte sul forum e in [5] sopperiscono a queste mancanze utilizzando tecniche molto particolari basate su elettronica aggiunta all'esterno del chip. Ad esempio è possibile generare audio di buona qualità utilizzando un segnale PWM e un filtro RC. Il firmware acquisisce così un'importanza considerevole nell'implementazione di periferiche 'in software' (esistono ad esempio interfacce per I^2C). È

chiaro però che usare il software per queste applicazioni può significare dedicare un Cog a questo compito.

3.3.2 Algoritmo su matrice con 2 Cog e VGA (Spin)

Questo esempio usa 3 Cog e una matrice di *word*. Il funzionamento è mostrato nello schema di figura 8. Due Cog colorano contemporaneamente la matrice in memoria: uno da giallo a rosso e viceversa, l'altro da giallo a blu e viceversa. Un Cog è adibito alla visualizzazione della su VGA (tramite un oggetto Spin) che appare come una scacchiera. Il risultato è che le celle gialle vengono periodicamente colorate di rosso e di blu con velocità diverse.

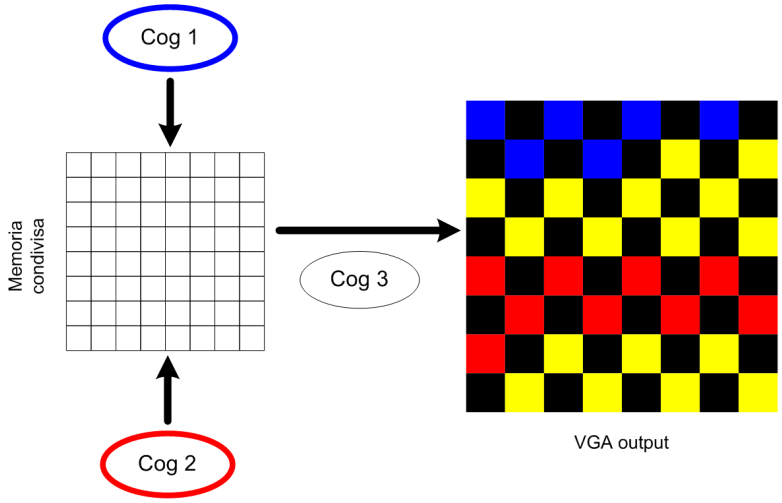


Figura 8: Schema esempio 1.

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  tiles    = vga#xtiles * vga#ytiles
  tiles32  = tiles * 16
OBJ
  vga : "vga_640x240_bitmap"
VAR
  long  sync, pixels[tiles32]
  long  stack1[10]
  long  stack2[3]
  word  colors[tiles]
  byte  nc1
  byte  nc2

PUB start | h, i, j, k, x, y

  'start vga
```

```
  vga.start(16, @colors, @pixels, @sync)
  initScr
  waitcnt(clkfreq * 2 + cnt)
  nc1 := cognew(compute, @stack1)
  compute2

PRI initScr | t,x,y,s

  t := 243
  s := 0
  repeat y from 0 to 14 step 1
    s:=1 - s
    repeat x from s to vga#xtiles-1 step 2
      colors[x + y*vga#xtiles] := t

PUB compute | t,x,y

  t:=243

  repeat
    repeat y from 7 to 0 step 1
      repeat x from 0 to vga#xtiles-1 step 1
        waitcnt(300000 + cnt)
        if (colors[x + y*vga#xtiles] == t)
            colors[x + y*vga#xtiles] := 255-t
        elseif (colors[x + y*vga#xtiles] == 255-t)
            colors[x + y*vga#xtiles] := t
        else
            colors[x + y*vga#xtiles] := 0

PUB compute2 | x,y,cb,c1,c2,j,k

  cb := 0
  c1 := 243
  c2 := 192

  repeat
    repeat y from 8 to 14 step 1
      repeat x from 0 to vga#xtiles-1 step 1
        waitcnt(500000 + cnt)
        if (colors[x + y*vga#xtiles] == c1)
            colors[x + y*vga#xtiles] := c2
        elseif (colors[x + y*vga#xtiles] == c2)
            colors[x + y*vga#xtiles] := c1
        else
            colors[x + y*vga#xtiles] := cb
```

3.3.3 Conversione AD e DA

Questo esempio, ripreso dal forum di Parallax, mostra come utilizzare un timer per campionare suoni dal microfono della demo board e riprodurli sull'amplificatiore audio[5]. Non esistono periferiche per la conversione AD/DA sul Propeller: tutto viene

[5] L'amplificatore presente sulla demo board è un componente MAX4411.

assolto da PWM generate dai timer alle quali vengono applicati filtri analogici esterni al Chip.
Nel caso specifico della campionatura dal microfono della demo board (convertitore AD), si utilizza una modulazione sigma-delta, il cui concetto è mostrato in figura 9. L'idea di base è di 'inseguire' un segnale $U_{in}(t)$ con un secondo segnale $U_r(t)$ prodotto da una PWM attraverso un filtro RC passa-basso. I due segnali vengono sottratti calcolando $E(t)$ che deve essere mantenuto entro una soglia limite. Variando il duty cycle della PWM, si cambia il valore di $U_r(t)$ e questa variazione è proporzionale a $U_{in}(t)$. Tutto questo viene prodotto dal timer A, usando le impostazioni mostrate nell'esempio. In particolare si noti la modalità di *CNTRMODE*: *POS detect w. feedback* (v. [2] e [4]).
Il timer A, una volta impostato, esegue queste operazioni in modo indipendente dal funzionamento del Cog. Una temporizzazione (v. sezione 3.2.4) permette poi di campionare il valore del registro di accumulazione ad intervalli regolari e di salvarlo in memoria. La risulozione del capionamento dipende dalla frequenza di lettura dell'accumulatore del timer A (temporizzazione). Questa frequenza è determinata dal valore del parametro *bits*, dipendente dalla frequenza di clock del timer (PLL). Le diverse frequenze di campionamento a 80MHz sono riportati in tabella 1. Per mantenere una frequenza di campionamento di 40kHz con diverse frequenze di clock (PLL) valgono le indicazioni della tabella 2.
La conversione DA è invece prodotta dal timer B con una PWM filtrata da un circuito RC. La modalità *DUTY_DIFFERENTIAL* consente di modificare la durata del duty cycle del timer B e di conseguenza permette di discriminare un valore di tensione tra 0 e U_{dd} (cioè la tensione di alimentazione, in questo caso 3.3V). Questo segnale viene poi amplificato dal componente MAX4411 sulla board e può essere ascoltato collegando semplici cuffie o casse. Da notare che il segnale d'uscita è stereo per cui utilizza due canali (pin 10 e 11).

Sample bits	Sample rate
5	2.5 MHz
6	1.25 MHz
7	625 KHz
8	313 KHz
9	156 KHz
10	78 KHz
11	39 KHz
12	19.5 KHz
13	9.77 KHz
14	4.88 KHz

Tabella 1: Risoluzioni di conversione AD/DA.

Clock	PLL	Risoluzione
80 MHz	16	11 bit
40 MHz	8	10 bit
20 MHz	4	8 bit
10 MHz	2	7 bit

Tabella 2: Impostazioni per campionamento a 40kHz.

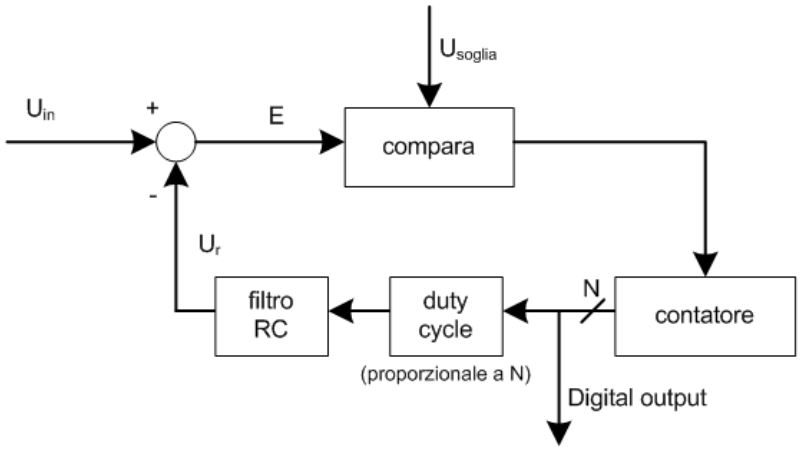


Figura 9: Sigma-Delta ADC.

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

  bits       = 11
  period  = 50_000
  mema    = 0                  'base address to store data in memory
  maxa    = 128

VAR
  long samples[256]

PUB Main
  cognew(@asm_record,@samples)

DAT
  org

asm_record  mov  dira,asm_dira          ' pins 8 (ADC) e 0 (DAC) outputs
                                        ' POS W/FEEDBACK mode per CTRA
            movs    ctra,#8             ' 08..00 cntra => 1000 => APIN
            movd    ctra,#9             ' 17..09 cntra => 1001 => BPIN
            movi    ctra,#%01001_000    ' 31..23 cntra => 01001_000
                                        ' => CNTRMODE = POS detect w. feedback
                                        ' => PLLDIV = VCO : 128
```

```
            movs    ctrb,#10            ' DUTY DIFFERENTIAL mode per CTRB
            movd    ctrb,#11
            movi    ctrb,#%00111_000

            mov     frqa,#1
            mov     asm_cnt,cnt         'prepare for WAITCNT loop
            add     asm_cnt,asm_cycles
            mov     daddr, PAR

:loop       waitcnt asm_cnt, asm_cycles ' sincronizza campionamento
            mov     asm_sample,phsa     ' valore attuale timer

            call    #asm_filter         ' applica filtro

            wrlong  asm_sample, #daddr  ' salva campione in memoria
            rdlong  asm_sample, #daddr

            call    #asm_play1          ' riproduci audio

            add     daddr, #4           ' incrementa indirizzo mem.
            mov     diff, #maxa
            sub     diff, daddr  wz
      if_z  mov     daddr, PAR

            jmp     #:loop               ' prossimo campionamento


asm_filter  sub     asm_sample,asm_old   ' filtro
            add     asm_old,asm_sample
            add     asm_old,asm_sample
asm_filter_ret  ret


asm_play1   shl     asm_sample,#32-bits   'produce output su FRQB
            mov     frqb,asm_sample
asm_play1_ret  ret

' Data
asm_cycles    long      |< bits - 1       ' sample time
asm_dira      long      $00000E02         ' output mask
asm_cnt       res       1
asm_old       long      1
asm_sample    long      1
daddr         res       1
diff          res       1
```

A Propeller Font Set

Nella ROM del Propeller si trova una serie di celle che compongono un set di caratteri (*font*) che può essere utilizzato per scrivere su VGA o uscita TV. I simboli che sono contenuti in memoria sono gli stessi del *font* utilizzato dal tool di sviluppo del Propeller, sia per quanto riguarda la programmazione che per quanto riguarda i commenti nei progetti.
Il set di caratteri Propeller è stato infatti sviluppato per semplificare e rendere più efficace il lavoro di documentazione di codici e oggetti Spin. Il set, come mostrato nella figura 10, non comprende solo l'alfabeto minuscolo e maiuscolo, numeri e punteggiatura, ma include anche una serie di simboli utili a disegnare schemi elettrici e formule matematiche. Questo può avvenire sia su schermo, tramite le periferiche VGA integrate nei Cog, sia nel tool di sviluppo. Nella sezione di documentazione infatti è possibile utilizzare i caratteri speciali per descrivere al meglio il proprio codice.
Il sistema di documentazione nel tool di sviluppo permette anche di nascondere automaticamente il codice e visualizzare solo i commenti, producendo quindi un documento che effettivamente illustra le informazioni sul codice che lo sviluppatore intende presentare. Queste tecniche sono molto utilizzate negli esempi di [5].

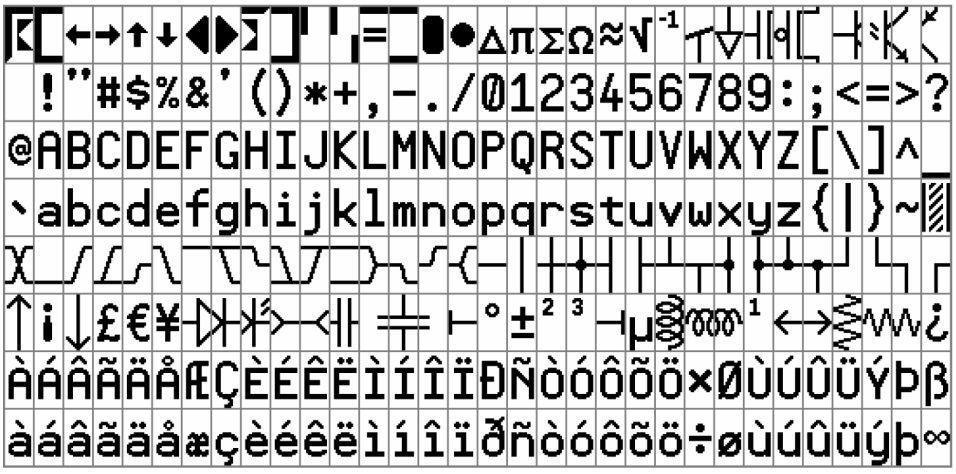


Figura 10: Il *Propeller Font Set* [3].

B Le tavole di funzioni matematiche

Come descritto in sezione 2.3, nel segmento ROM della memoria, si trovano dei valori tabellati di alcune funzioni quali: seno, logaritmo e anti-logaritmo. I valori sono calcolati per argomenti equidistanti (ad esempio, gli argomenti del seno sono angoli presi ad ogni 0.0439°) e memorizzati in locazioni adiacenti e progressive.
L'accesso e l'uso dei valori memorizzati nelle tavole dipende, ovviamente, dall'applicazione e dalla funzione rappresentata. L'appendice B di [2] riporta numerosi esempi assembly per accedere alle memorie. Di seguito verrà riportato l'uso della tabella del seno in grado di generare anche il coseno per un dato angolo.
Fisicamente sono rappresentati 2048 immagini di un quadrante (angoli da 0° a 90°), con una risoluzione di 0.0439°. Su [2] e su [3] è riportato che la tavola ha 2049 punti. Questo è vero se si considera che l'immagine del punto 2049 è identica a quella del primo punto del quadrante successivo. I valori sono memorizzati in locazioni da 16 bit per questioni di praticità di accesso. Nella sezione B.1 è riportata la spiegazione delle caratteristiche di questa implementazione fornita dagli sviluppatori Parallax.

B.0.4 Esempio di calcolo di $sin(x)$

quadrante:	1	2	3	4
angolo:	$0000..$07FF	$0800..$0FFF	$1000..$17FF	$1800..$1FFF
indice tabella:	$0000..$07FF	$0800..$0001	$0000..$07FF	$0800..$0001
mirror:	+offset	-offset	+offset	-offset
flip:	+sample	+sample	-sample	-sample

```
' Get sine/cosine
'
' Tavola seno: $0000..$07FF
' argomento: sin
' in : sin[12..0] contiente l'angolo (0° a 360°)
' out: sin contiene il valore del segnato (sin/cos)
' compreso tra $0000FFFF ('1') e $FFFF0001 ('-1')
'
getcos      add     sin,sin_90        'for cosine, add 90°
getsin      test    sin,sin_90 wc    'get quadrant 2|4 into c
            test    sin_sin_180 wz    'get quadrant 3|4 into nz
            negc    sin,sin           'if quadrant 2|4, negate offset
            or      sin,sin_table     'or in sin table address >> 1
            shl     sin,#1            'shift left to get final word address
            rdword  sin,sin           'read word sample from $E000 to $F000
            negnz   sin,sin           'if quadrant 3|4, negate sample
getsin_ret
getcos_ret  ret  '39..54 clocks
                                      '(variance due to HUB sync on RDWORD)
            sin_90 long $0800
            sin_180 long $1000
            sin_table long $E000 >> 1 'sine table base shifted right
            sin long 0
```

B.1 Risposte dal forum

Posted 4/13/2007 9:13 AM Hi everybody,
I've some doubts on the sine table.
The values of the sine for angles between 0° and 90° are divided into 2049 samples. Now, for each (unsigned) sample 12 bits should be enough, but 16 bits are used instead.
The questions are:
why 16 bits?
with 16 bits we can have 65536 values, but as we understand we are just using 2049 of them. How are the 2049 values choosen? What are the advantages? Are we completely missing the point?
The function getsin (or getcos) reported on the page 423 of the manual takes as input an angle from 0° to 360°.
The question is: should I use a 14-bits values to express the angle? (4 x 2049 = 8196 => 14 bits)
If I want to generate a sine-wave signal using the audio amplifier of the demo-board, is it enough to change the freqb register of a timer set as follows within a certain sampling period?

```
movs  ctrb,#10
movd  ctrb,#11
movi  ctrb,#%00111_000
```

Thanks,
IB.

Mike Green Posted 4/13/2007 9:41 AM The range 0 to 90 degrees is divided into 2048 equal intervals. The table includes both endpoints because it's is used in reverse for the cosine and the extra point makes this easier. Memory comes in units of 8 bits, so 16 bits is as easy to provide as 12 bits and gives us a little more accuracy. You actually need 13 bits to represent the angle from 0 to 359.9999.... degrees. There are only 2048 values per quadrant since the 2049th value is the same as the first value of the next quadrant, thus you have

$$4 \cdot 2048 = 8192$$

which requires 13 bits.

Mike Green Posted 4/13/2007 9:54 AM Tracy Allen posted a sinewave generator some time ago that just needs a little filtering to produce fairly clean sine waves. I can't find the link, but here is a copy of it.
File Attachment: Frequency Generator.zip 8KB (application/zip)

Tracy Allen Posted 4/13/2007 10:13 AM The values in the table are computed for the table with 2^{16} possible values for y:

$$y = 65535 \cdot sin(\theta)$$

for $\theta = 0$ to 90 degrees, 2048 equal intervals with 2049 end points.
That gives better precision for y than would be had from 12 bits,

$$y' = 4096 \cdot sin(\theta)$$

Yes you enter with 2^{13} bits and use the top two bits for the quadrant, and 11 bits to index into the table.
It is possible to enter with more bits and interpolate. For example, if you enter with 16 bits to describe a circle, then bits 14, 15 would be the quadrant, bit 3 to 13 would index into the table, and bits 0 to 2 could be used for interpolation.
Here is the link to a tutorial that shows specifically how to generate a sine wave output using one counter for phase and one counter for amplitude.

Chip Gracey (Parallax) Posted 4/13/2007 10:28 AM The sine table gives 8,192 16-bit samples for a complete circle. It must be read in the following order for a complete circle:
sample[0 to 2047]
sample[2048 to 1]
-sample[0 to 2047]
-sample[2048 to 1]
The funny overlaps are to ensure minimal and symmetrical 0 and 180 degree points and maximal and symmetrical 90 and 270 degree points.
To output sine waves, an RC integrator must be driven with delta-modulated (CTR mode) sine samples. Attached is a program that generates sine wave outputs on pins 10 and 11 and sweeps their frequencies at different rates.

Riferimenti bibliografici

[1] `http://forums.parallax.com`

[2] *Propeller Manual*, Parallax, 2006

[3] *Propeller P8X32A Preliminary Datasheet*, rev. 0.2, Parallax inc., 2007

[4] *AN001 - Propeller Counters*, v. 1.0, Parallax inc., 12.11.2006

[5] Charlie Johnson, *Spin Code Examples*, Parallax inc., 2006-7

[6] Phil Pilgrim, *Propeller Tricks & Traps*, Bueno Systems inc., 03.06.2006