

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE
Corso di Laurea Magistrale in Ingegneria dell'Energia Elettrica

Tesi di Laurea Magistrale

**Aggiornamento del firmware di controllo
di un sistema WPT per l'impiego
di un microcontrollore dual-core**

Relatore:
Professore
Manuele Bertoluzzo

Laureando:
Fabio Baratella
Matricola 1242072

Anno Accademico 2021-2022

Indice

Introduzione	i
1 Presentazione sistema e inizializzazione	1
1.1 Sistema utilizzato	1
1.1.1 Microcontrollore TMS320F28379D	2
1.1.2 Scheda Controllo e Potenza	3
1.1.2.1 Controllo	3
1.1.2.2 Potenza	4
1.1.3 Strumenti alimentazione e misura	5
1.2 Programma e file	6
1.2.1 Main	8
1.2.2 Interrupt Service Routine	10
1.2.3 Algoritmo di controllo	13
1.2.4 Hardware Interface	13
1.3 Inizializzazione microcontrollore e delle periferiche	14
1.3.1 Inizializzazione hardware	15
1.3.2 Funzioni richiamate	15
1.3.3 Direttiva <code>#define</code>	19
2 Moduli utilizzati	21
2.1 Clocking	21
2.2 Timer	23
2.3 Peripheral Interrupts	24
2.4 GPIO	27
2.4.1 Registri di controllo	27
2.4.2 Muxing periferiche	29
2.4.3 Qualificazione del segnale	30
2.5 Crossbar (X-BAR)	30
2.5.1 Input X-BAR	30
2.5.1.1 External Interrupt	32
2.6 Enhanced Pulse Width Modulator	34

2.6.1	TB	36
2.6.2	CMP	37
2.6.3	AQ	38
2.6.4	DB	38
2.6.5	ET	41
2.6.6	TZ	41
2.6.7	DC	43
2.6.8	Configurazione ePWM	44
2.7	Analog to Digital Converter	47
2.7.1	Configurazioni	50
2.7.2	SOC	51
2.7.3	Sincronizzazioni tra moduli	52
2.8	SPI e SCI	53
2.8.1	Serial Peripheral Interface	53
2.8.1.1	Esempio trasmissione dati	55
2.8.2	Serial Communications Interface	55
2.8.2.1	Architettura SCI	56
2.8.2.2	Struttura del frame	57
3	Aggiornamento software	59
3.1	High Resolution PWM	59
3.1.1	Configurazione	60
3.1.2	CMPHR	62
3.1.3	TBPRDHR e TBPHSHR	66
3.1.3.1	Limiti	68
3.1.3.2	Problemi riscontrati	68
3.2	Interprocessor Communication	70
3.2.1	Esempio uso IPC	70
3.2.2	Variabile in GS0 RAM condivisa	72
3.3	Connessione RS232	74
3.3.1	Software Termite	75
3.4	CCS debug software dual core	76
3.5	Scheda di interfaccia	78
3.6	Aggiornamenti futuri	78
4	Funzionamento Dual Core	81
4.1	Main CPU2	81
4.2	RS232 management	82
4.2.1	Registri IPC	85
4.2.2	Esempio scambio dati	86
4.3	Modalità standalone	87

INDICE	5
4.4 Prestazioni	89
4.4.1 While	89
4.4.2 Interrupt ADC	90
4.4.3 Confronto microcontrollori	92
4.5 Collegamento al sistema completo	95
4.5.1 Strumenti	95
4.5.2 Risultati	96
A Schemi completi	99
Conclusioni	103
Bibliografia	105

Introduzione

Lo scopo della tesi è aggiornare il firmware di controllo per un sistema *Wireless Power Transfer* pensato per la ricarica di veicoli elettrici. L'azione si è resa necessaria dal momento in cui è stato richiesto un cambio di microcontrollore: le potenzialità fornite dal LAUNCHXL-F28379D garantiscono una maggiore capacità di calcolo, la possibilità di un funzionamento dual core e un numero superiore di periferiche. Il precedente software si basa sul vecchio MCU (*microcontroller unit*), appartenente alla stessa famiglia di prodotti della Texas Instruments. Il fratellino minore tuttavia presenta dei limiti per quanto concerne *clock* e funzionalità. Un *upgrade* hardware è stato dunque doveroso.

Cambiando il cervello che gestisce la potenza e la comunicazione tra le varie parti del sistema si è dovuto tuttavia procedere ad una revisione puntigliosa del precedente software. I registri e le configurazioni effettuate nel passato infatti non garantiscono la compatibilità con il nuovo microcontrollore, che presenta in alcune periferiche un'architettura del tutto nuova.

La divisione di questo documento si sviluppa con l'obiettivo di sviscerare il funzionamento del sistema già esistente, scoprire le principali caratteristiche del chip TMS320F28379D ed implementarle correttamente. Il primo intento è quello di ottenere le stesse funzionalità e verificare il regolare esercizio del sistema. Il passo successivo sarà quello di addentrarci nelle potenzialità dual core e fornire miglioramenti laddove possibile con il numero maggiore di periferiche.

I capitoli presenti seguiranno quindi questo sviluppo. Il primo servirà come *Presentazione sistema e inizializzazione*: partiamo dagli strumenti presenti sul banco di lavoro, fino ad arrivare alle prime function che inizializzano l'hardware, passando per una breve presentazione dei file presenti. Il secondo capitolo illustrerà nel dettaglio alcuni tra i *Moduli utilizzati*, facendo maggiormente attenzione alle nuove caratteristiche o alle peculiarità dell'architettura più recente. Il terzo capitolo si concentra sull'*Aggiornamento software*, in particolare sulla pwm ad alta risoluzione, al modulo IPC per la comunicazione tra i due core ed infine la comunicazione seriale. Il quarto ed ultimo capitolo

cerca di illustrare il *Funzionamento Dual Core* implementato, ponendo poi l'attenzione su un confronto con il precedente F28335.

Durante questa avventura i compagni di viaggio più fedeli sono sicuramente il *Technical Reference Manual* [1], che invito a consultare ogni volta che si palesa un dubbio, e le librerie C2000 fornite da Texas Instruments, in cui sono presenti file di ogni tipo necessari al corretto funzionamento del microcontrollore e ricchi di esempi operativi.

In conclusione a questa prolissa introduzione, un ultimo appunto: durante la spiegazione ho trovato particolarmente utile riportare direttamente i pezzi di codice presenti all'interno del progetto per una migliore comprensione delle operazioni eseguite. Questi tuttavia sono delle estrapolazioni che cercano di riassumere le parti più importanti e non contengono certamente ogni passaggio. A questo scopo nella descrizione del *Code* è riportato il file a cui la citazione fa riferimento; il numero delle righe a lato servono per orientarsi tra le centinaia presenti, potrebbero tuttavia non corrispondere precisamente date le numerose versioni e tentativi di firmware realizzati.

Capitolo 1

Presentazione sistema e inizializzazione

Questo primo capitolo sarà diviso in due parti principali. La *presentazione del sistema*, che comprende una rapida illustrazione del microcontrollore, delle schede di potenza, controllo e dell'oscilloscopio, è utile per poter comprendere la configurazione sul banco di lavoro. La seconda è fondamentale per muoversi all'interno del *programma*, che racchiude molteplici file *.c*, *.h* e quelli necessari al funzionamento del microprocessore; l'*inizializzazione* non è altro che la configurazione dei registri che agiscono sulle diverse periferiche.

1.1 Sistema utilizzato

Il MCU (micro-controller unit) è collegato al PC tramite porta USB, che garantisce l'alimentazione e lo scambio di dati per il debug. È possibile, come spiegato nel paragrafo 5.2 *Power Domain, LAUNCHXL-F28379D Overview* [8], fornire +3.3V e 5V esternamente, e garantire l'isolamento della scheda tramite JP1, JP2, JP3 (vedere Fig 1.2a).

I collegamenti dei vari pin (Fig 1.1) sono stati effettuati attraverso dei *jumper wires*: nella Fig 1.4 si possono notare i GPIO utilizzati, che verranno ripresi nel capitolo successivo.

Una volta alimentata la scheda di controllo e di potenza, le diverse forme d'onda sono state visualizzate con delle sonde e un oscilloscopio a quattro canali.

1.1.1 Microcontrollore TMS320F28379D

Il microcontrollore TMS320F28379D della Texas Instruments serie C2000 ha una CPU Dual Core 32 bit floating point con clock di 200 MHz. Le principali caratteristiche sono:

- due co-processor CLA programmabili indipendentemente;
- 1MB Flash (512KB per core) e 204KB RAM (102KB per core);
- molteplici opzioni di connettività: CAN, I2C, SPI, SCI, USB, McBSP;
- 16 HRPWM output con controllo edge 150ps;
- 3 QEP input;
- 4 ADC con funzionamento 12-bit/16-bit;
- 3 DAC output con buffer 12-bit;
- 2 connettori 40-pin per BoosterPack;
- connettore High Density per connettività EMIF ed espansione IO sul lato inferiore (J9).

Il Development Kit, LAUNCHXL-F28379D, fornisce un totale di 80 pin e la possibilità di connettere dei BoosterPack. La Fig 1.1, che illustra questo tipo di collegamenti, è utile per comprendere la posizione dei GPIO e parte delle loro funzioni; per scoprire tutte le possibili connessioni è fondamentale fare uso del *Technical Reference Manual* [1]; in particolare al paragrafo 8.7 *GPIO and Peripheral Muxing* sono presenti tutte le combinazioni.

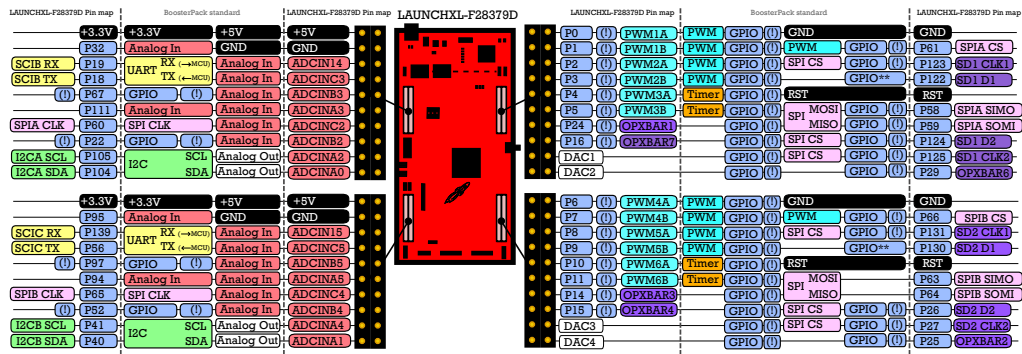


Figura 1.1. Pin disponibili

In Fig 1.2, inoltre, si può notare la serigrafia degli strati superiore ed inferiore; nel *bottom* è importante la nomenclatura dei pin, alla quale faremo riferimento quando parleremo dei GPIO.

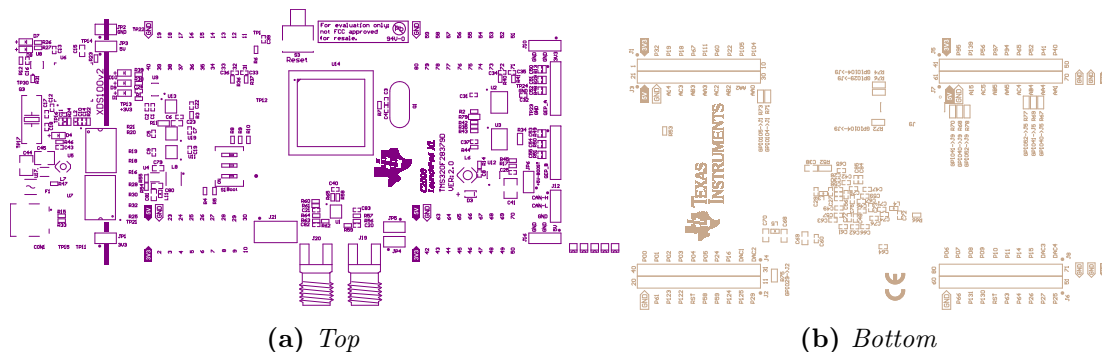


Figura 1.2. Serigrafia

1.1.2 Scheda Controllo e Potenza

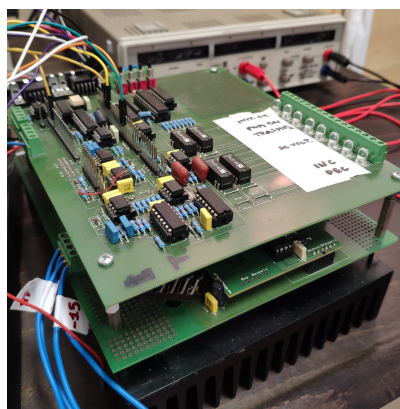


Figura 1.3

Lo scopo delle schede è la gestione di potenza per un sistema *Wireless Power Transfer WPT* [11]. Come si può vedere in Fig 1.3, le due parti sono sovrapposte: nella board superiore giace la scheda di controllo, in quella inferiore sono presenti gli switch ad alta frequenza.

I collegamenti ai pin del micro-controllore sono effettuati tramite dei cavetti. Nei successivi paragrafi si cercherà di illustrare in modo rapido le principali caratteristiche di entrambe le board.

Lo schema completo della scheda di *controllo* si trova nell'Appendice A, in Fig A.1; il circuito di *potenza* in Fig A.2, grazie al quale è possibile capire quali sono i collegamenti tra

i vari componenti nel pcb.

1.1.2.1 Controllo

I pin si trovano nella parte laterale e garantiscono la comunicazione con le varie periferiche della scheda. Il collegamento completo dei cavi jumper è riportato in Fig 1.4; anche se confusionario, questo tipo di approccio è stato necessario dal momento che la disposizione dei pin è totalmente diversa

rispetto alla scheda di sviluppo del microcontrollore precedente. Gli ultimi elementi degni di nota sono ovviamente i quattro pulsanti e i vari led che costituiscono l'interfaccia uomo-macchina.

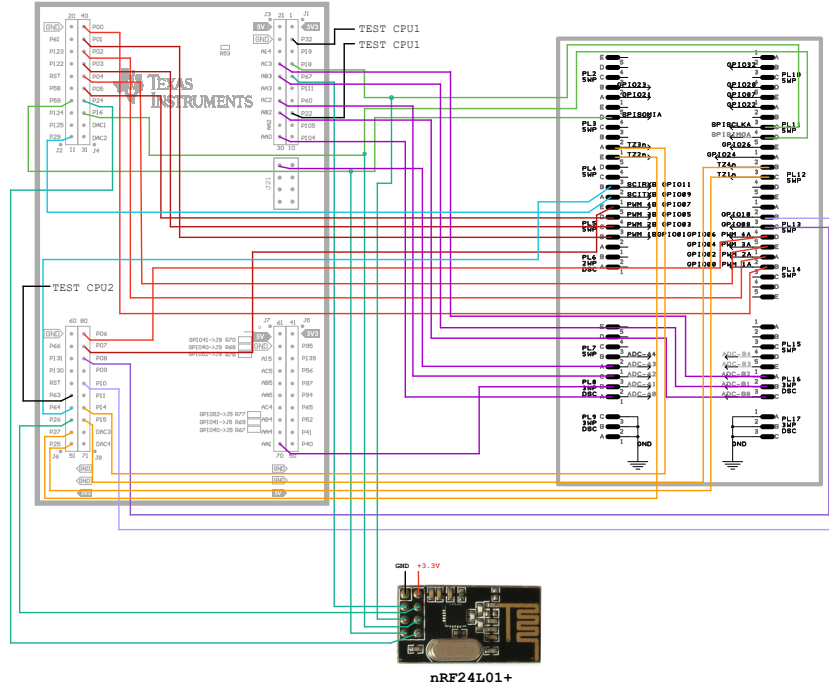


Figura 1.4. Collegamenti tra retro del *microcontrollore* (sinistra), pin *scheda di controllo* (destra), pin *modulo RF* (in basso)

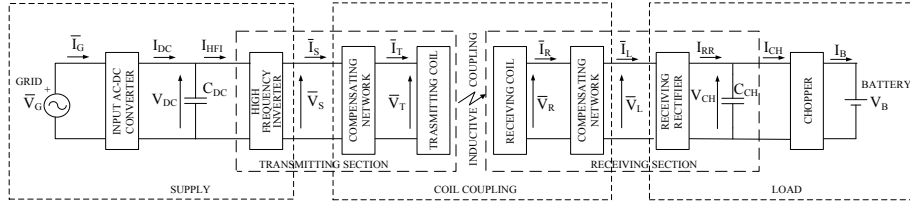
1.1.2.2 Potenza

Il *sistema WPT* (Fig 1.5a) sfrutta l'accoppiamento induttivo tra due bobine. L'alimentazione lato trasmettente è realizzata tramite un *HFI* (Fig 1.5b) che fornisce delle onde quadre tramite le due gambe A e B, sfasate tra loro dell'angolo Φ (Fig 1.5c). La scheda di potenza serve a realizzare tale inverter ad alta frequenza.

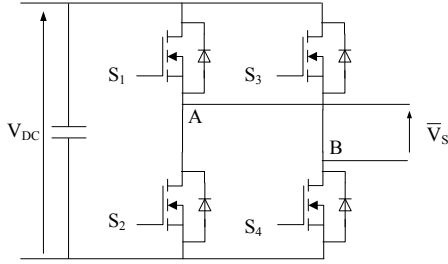
La *phase shift technique* utilizza l'angolo Φ per gestire l'ampiezza della *fondamentale* della tensione alternata in uscita, che risulta:

$$V_s = \frac{4}{\pi} \cdot V_{DC} \sin\left(\frac{\Phi}{2}\right) \quad (1.1)$$

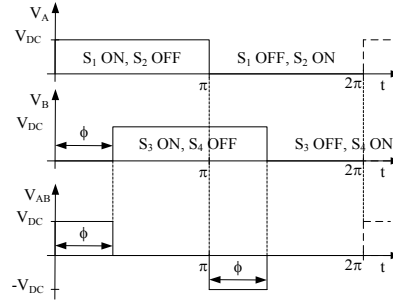
La tensione del bus DC è fornita dall'esterno e per le prove nel banco di lavoro è stata scelta 15V; nel sistema pienamente funzionante V_{DC} è fornita da un ponte a diodi collegato alla rete e risulta di 365V.



(a) Schema generale accoppiamento induttivo



(b) High frequency inverter



(c) Forme d'onda

Figura 1.5. Wireless power transfer

1.1.3 Strumenti alimentazione e misura

Per fornire tensione alle schede è stato utilizzato un alimentatore *Hameg, Triple Power Supply HM 7042* [15] in grado di fornire tre tensioni DC indipendenti; le specifiche sono riportate in Tab 1.1. La scheda di potenza è alimentata con tensioni di $+15V$, $-15V$ e $+5V$.

Per le misure delle forme d'onda è stato usato l'oscilloscopio *Agilent Technologies, MSO6034A*: 300MHz, 2GSa/s, è dotato di 4 canali analogici e 16 digitali. I grafici che si vedranno nel documento sono estratti da tale strumento ed elaborati poi con Matlab.

Tabella 1.1. Triple Power Supply HM7042

Tensioni	2x 0-32V	2.7V – 5.5V ($\pm 0.1V$)
Risoluzione	100mV	10mV
Corrente output	2x 0-2A	0-3.3A
Setting range	2x $< 0.05-2A$	0.05-3.3A
Risoluzione	1mA (0-1999mA), 10mA ($\geq 2A$)	10mA



(a) HM7042

(b) MSO6034A

Figura 1.6. Alimentatore ed oscilloscopio

1.2 Programma e file

La programmazione del MCU avviene tramite il software *Code Composer Studio* [13] e le librerie *C2000Ware* [14] disponibili nel sito della Texas Instruments. Una volta installato il programma, è necessario importare il progetto alla voce **Project** ▶ **Import CCS Project**. Se dovessero esserci problemi di connessione con il F28379D, potrebbe bastare creare un nuovo *Target Configuration File* (tasto destro sul progetto, e poi New). La connessione avviene tramite XDS100v2 USB Debug Probe, ed è possibile verificarla anche andando nelle **Properties** ▶ **General** ▶ **Project** e cliccare su *Verify*.

La versione del compilatore è *TI v21.6.0.LTS*; il file utilizzato dal linker riportato nel progetto *2837xD_RAM_lnk_cpu1.cmd* si trova all'interno delle librerie e materiale di supporto C2000, il percorso è **C:** ▶ **ti** ▶ **c2000** ▶ **C2000Ware_4_01_00_00** ▶ **device_support** ▶ **f2837xd** ▶ **common** ▶ **cmd** se si esegue l'installazione consigliata.

Si prosegue poi con l'inserimento degli *Includes*, librerie disponibili nelle cartelle **C:** ▶ **ti** ▶ **c2000** ▶ **C2000Ware_4_01_00_00** ▶ **device_support** ▶ **f2837xd** ▶ **common** ▶ **include**, e la successiva **...** ▶ **device_support** ▶ **f2837xd** ▶ **headers** ▶ **include**. Infine, per la calibrazione del modulo ad alta risoluzione per la pwm inseriamo **...** ▶ **libraries** ▶ **calibration** ▶ **hrpwm** ▶ **f2837xd** ▶ **include**.

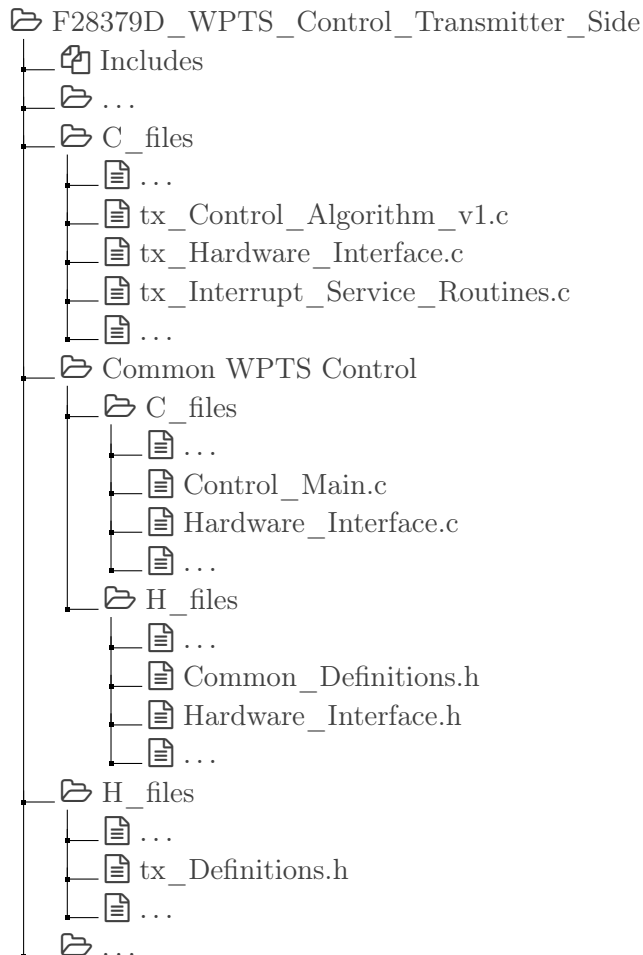
Il passaggio successivo riguarda l'inserimento di alcuni *Predefined Symbols*, necessari alla configurazione iniziale: con il tasto destro sul progetto, si accede **Properties** ▶ **Build** ▶ **C2000 Compiler** ▶ **Predefined Symbols** e bisogna inserire "CPU1" e "_LAUNCHXL_F28379D".

In conclusione di questa prima configurazione iniziale, si devono aggiungere delle librerie per il corretto funzionamento del modulo MEP, fondamentale nella pwm ad alta risoluzione (vedere capitolo successivo 3.1). Il file *SFO_v8_fpu_lib_build_c28.lib* viene quindi inserito dalla directory **C2000** **...** ▶ **libraries** ▶ **calibration** ▶ **hrpwm** ▶ **f2837xd** ▶ **lib**.

Ritengo ora imprescindibile illustrare le diverse directory presenti all'interno del Project, contenenti ciascuna numerosi file. È importante riconosce i più importanti, e al loro interno quali function sono richiamate durante l'attività del micro. Il lettore scoprirà presto quanto sia facile perdersi tra le centinaia di righe di codice: i richiami tra variabili, costanti, funzioni e interrupt sono onnipresenti tra i molteplici file.

Il codice originale, da cui parte poi l'adattamento, fa riferimento al vecchio micro-controllore F28335. Non tutti i file sono stati tuttavia modificati; i più significativi sono gli "Hardware_Interface" che, come spiegato più avanti, contengono la maggior parte delle configurazioni riguardanti il micro. Il *main* al contrario, non è stato quasi per nulla aggiornato in quanto non è stato necessario.

In seguito si può trovare un *directory tree* con i file più importanti.



Vediamo ora i file fondamentali per capire il lavoro del micro.

1.2.1 Main

All'interno di `Project▶Common WPTS Control▶C_files` troviamo il file *Control_Main.c*, che è il programma principale. L'estratto di Code 1.1 che segue serve a capire il funzionamento approssimativo e verranno descritte solo le funzioni principali, evidenziate in rosso.

Dopo le linee iniziali `#include`, `extern int` e `int function()`, il programma inizia con il ciclo `while(1){...}`, condizione che è sempre verificata e continua finché il micro rimane alimentato.

La seconda istruzione di controllo condizionale `if` (riga 99) verifica che la variabile `Time_Interval_of_1ms_Elapsed` sia posta al valore `TRUE` (cioè la condizione *vera* `1==1`); questo accade esattamente ogni *1ms*: all'interno dell'*CPU_Timer0_isr* (*interrupt service routine*), eseguita ogni *250μs*, la variabile `Time_Interval_Count` viene incrementata finché, arrivando al valore 4, pone la condizione di cui sopra. Gli interrupt, e in particolare quello del timer, verranno ripresi nella Sezione 1.2.2.

Code 1.1. Control_Main.c

```

93 while(1){
94
95     if(nRF24L01p_has_been_Initialized==TRUE){
96         Manage_WPTS_Communications(&Received_New_RF_Data);
97     }
98
99     if(Time_Interval_of_1ms_Elapsed==TRUE)
100     {
101         Time_Interval_of_1ms_Elapsed=FALSE;
102         if(MCP23S17_has_been_Initialized==TRUE){
103             ...
104         }
105
106     switch(State_of_the_Management_Algorithm)
107     {
108         case STATE_MACHINE_INITIALIZATION:
109             Result&=Initialize_the_Software();
110             Result&=Initialize_the_Hardware();
111
112             if(Result==TRUE){
113                 State_of_the_Management_Algorithm=PARAMETERS_INITIALIZATION;
114                 Previous_State_of_the_Management_Algorithm=STATE_MACHINE_INITIALIZATION;
115                 Remaining_Duration_of_the_Current_State=PARAMETERS_INITIALIZATION_DURATION;
116             }
117             if(Remaining_Duration_of_the_Current_State==0){
118                 State_of_the_Management_Algorithm=INITIALIZATION_ALARM;
119                 Previous_State_of_the_Management_Algorithm=STATE_MACHINE_INITIALIZATION;
120                 Remaining_Duration_of_the_Current_State=INITIALIZATION_ALARM_DURATION;
121                 Time_Interval_of_1ms_Elapsed=FALSE;
122                 // If there is an error here may be timer is not correctly set
123             }
124             break;
125
126         case INITIALIZATION_ALARM:
127             if(Remaining_Duration_of_the_Current_State==0){

```

```

132     Blocking_Alarms_Record=INITIALIZATION_ALARM_MASK;
133     State_of_the_Management_Algorithm=IDLE_LOOP;
134     Previous_State_of_the_Management_Algorithm=INITIALIZATION_ALARM;
135     Remaining_Duration_of_the_Current_State=LOOP_DURATION;
136 }
137 break;
138
139     case PARAMETERS_INITIALIZATION:
140     ...
141     break;
142
143     case MCP23S17_AND_NRF24L01p_INITIALIZATION:
144     ...
145     break;
146
147     case ADC_OFFSET_COMPENSATION:
148     ...
149     break;
150
151     case WAITING_FOR_TURN_ON_PUSHBUTTON:
152     ...
153     break;
154
155     case OPERATIONAL:
156     ...
157     break;
158
159     case OPERATIONAL_ALARM:
160     ...
161     break;
162
163     case IDLE_LOOP:
164     break;
165
166 } //switch(State_of_the_Management_Algorithm)
167
168 } //if(Time_Interval_of_1ms_Elapsed==TRUE)
169 else{
170     Result=FALSE;
171 }
172 } //while(1)

```

All'interno del ciclo `if`, ripetuto dopo `1ms`, si sviluppa `switch`, istruzione di controllo condizionale fondamentale. I diversi stati dipendono da `State_of_the_Management_Algorithm` e si dividono in una prima parte di *inizializzazione*:

- `STATE_MACHINE_INITIALIZATION` la parte di inizializzazione del software e dell'hardware;
- `INITIALIZATION_ALARM` è possibile notare che nel caso in cui la variabile `Remaining_Duration_of_the_Current_State` arrivi a zero allora si entra nello stato di allarme (il micro deve essersi bloccato);
- `PARAMETERS_INITIALIZATION` vengono caricati parametri dalla EEPROM;

- `MCP23S17_AND_NRF24L01p_INITIALIZATION` si inizializzano il modulo SPI sulla scheda di controllo e quello in radio frequenza;

ed una seconda *operativa*:

- `ADC_OFFSET_COMPENSATION` calcola i valori di offset per le acquisizioni dell'ADC;
- `WAITING_FOR_TURN_ON_PUSHBUTTON` aspetta il comando in arrivo dal SPI che comunichi che il pulsante sia stato premuto;
- `OPERATIONAL` lo stato operativo è richiesto affinché all'interno dell'interrupt `ADC_isr` si possa procedere ad abilitare pwm con `function Enable_PWM()` e scrivere sui registri dei CMP e dei TBPRD;
- `OPERATIONAL_ALARM` allarme durante funzionamento.

Ad ogni stato, si passa al successivo solo se tutte le funzioni richiamate precedentemente hanno avuto successo (`if (Result==TRUE)`); in caso di errori si passa allo stato `ALARM`.

Infine le due funzioni evidenziate in rosso, `Initialize_the_Software()` e `Initialize_the_Hardware()`, sono importanti per quanto riguarda l'inizializzazione dell'hardware e verranno riprese nella Sezione 1.3.

1.2.2 Interrupt Service Routine

Un interrupt è un segnale che impone alla CPU di mettere in pausa l'esecuzione corrente del programma e proseguire in un diverso pezzo di codice noto come *interrupt service routine* (ISR). All'interno del Progetto ne vengono usati diversi; di seguito saranno trattati solo i più importanti.

Per la corretta impostazione di una isr si veda Sezione 1.3, dove sono presenti le varie inizializzazioni.

Gli interrupt, presenti nel file `tx_Interrupt_Service_Routines.c` all'interno di `Project\C_files`, sono:

1. `ADC_isr`: legato al ADC, parte del codice è riportato in Code 1.2;
2. `PWM TZ1_isr`: nel momento in cui è attivata questa isr del Trip Zone, i moduli pwm vengono disabilitati;
3. `SCI_isr`: due interrupt per i moduli A e B in ricezione Rx;
4. `Timer 0_isr`: un estratto è presente in Code 1.3, ed il suo funzionamento è legato al ciclo nel *main*;

5. RF transceiver isr: tramite interrupt esterno, segnala l'attesa del modulo RF.

Da notare che alla fine di ogni isr bisogna inserire un **CLEAR** sui bit dei Flag e del PIEACK.

Code 1.2. tx_Interrupt_Service_Routines.c

```

39 interrupt void ADC_isr(){
40
41  /* Acquisition of the ADC samples */
42  ...
43
44  /* Computation of ADC offsets */
69  if(State_of_the_Management_Algorithm==ADC_OFFSET_COMPENSATION){
70      ...
71  }
72  else{
73
74      /* Computation of acquired quantities */
75      ...
76
77      /* Check of Alarms */
78      ...
79
80      /* Set PWM */
197  Half_of_PWM_Period=HALF_OF_INVERTER_PWM_NOMINAL_PERIOD;
198  Quarter_of_PWM_Period=QUARTER_OF_INVERTER_PWM_NOMINAL_PERIOD;
199  tx_Control_Algorithm_v1();
200  //tx_Amplitude_Modulated_Supply_Voltage();
201
202  if((State_of_the_Management_Algorithm==OPERATIONAL) &&
203      (Blocking_Alarms_Record==0x0000))
204  {
205      if(PWM_Enabled==FALSE){
206          PWM_Enabled=Enable_PWM();
207      }
208      EALLOW;
209      #ifndef MANUAL_DUTY_CYCLE
210      Duty_Cycle=(int)(Half_of_PWM_Period*
211          Two_Times_Arcsin_over_PI_Interp(Supply_Voltage_Reference));
212      #else
213      if(External_Reference_1>(Half_of_PWM_Period)){
214          External_Reference_1=Half_of_PWM_Period;
215      }
216      if(External_Reference_1<0){
217          External_Reference_1=0;
218      }
219      Duty_Cycle=External_Reference_1;
220      #endif
221
222      PERIOD_PHASE_A=(int)Half_of_PWM_Period;
223      PERIOD_PHASE_B=(int)Half_of_PWM_Period;
224      PERIOD_AUX_PWM=(int)Half_of_PWM_Period;
225      DUTY...UP...A =(int)(Quarter_of_PWM... +Half_of_Duty...);
226      DUTY...DOWN...A=(int)(Quarter_of_PWM...-Half_of_Duty...);
227      DUTY...UP...B =(int)(Quarter_of_PWM... +Half_of_Duty...);
228      DUTY...DOWN...B=(int)(Quarter_of_PWM...-Half_of_Duty...);
229      DUTY_CYCLE_AUX_PWM_A=(int)Quarter_of_PWM_Period;
230      DUTY_CYCLE_AUX_PWM_B=(int)Half_of_PWM_Period;

```

```

231     EDIS;
232 }
233 else{
234     PWM_Enabled=(!Disable_PWM());
235 }
236 }
237 // Reinitializes ADC for next interrupt
238 RESET_THE_ADC_SAMPLING_SEQUENCE;
239 CLEAR_THE_ADC_INTERRUPT_FLAG;
240 CLEAR_THE_PIE_INTERRUPT_FLAG_OF_THE_ADC; // Acknowledges
241 }

```

All'interno della isr dell'ADC ci sono diverse cose da notare: oltre alle più ovvie operazioni legate all'acquisizione, verso riga 197 cominciano le impostazioni dei registri dei moduli PWM, che si dividono principalmente in TBPRD period, e CMP compare. Queste tuttavia agiscono solo se le condizioni `if` a riga 202, cioè lo stato `OPERATIONAL` e l'assenza di allarmi, sono rispettate.

Le espressioni evidenziate in rosso nel codice sovrastante sono:

- `tx_Control_Algorithm_v1()` richiamo all'algoritmo di controllo, e può essere sostituita con `tx_Amplitude_Modulated_Supply_Voltage()` che si trova alla linea successiva;
- l'abilitazione dei moduli pwm con `Enable_PWM()`;
- infine la variabile `Duty_Cycle` che, nel caso in cui sia definita la costante `MANUAL_DUTY_CYCLE`, è possibile impostare manualmente. In particolare, durante le prove sul banco, il duty cycle è stato posto uguale a `External_Reference_1`, un numero che aumenta o diminuisce indipendentemente dai pulsanti premuti sulla scheda di controllo. Al lato pratico, la variabile impone lo sfasamento Φ tra le due onde quadre che andranno a pilotare le due gambe dell'inverter ad alta frequenza, Fig 1.5c.

Il secondo ed ultimo interrupt presentato è quello del Timer0. L'approfondimento sul modulo del timer avverrà nel prossimo capitolo (2.2), ma le poche righe di codice che seguono rivelano le operazioni eseguite.

Code 1.3. tx_Interrupt_Service_Routines.c

```

341 interrupt void CPU_Timer0_isr(){
342     static int Time_Interval_Counter=0;
343     if(++Time_Interval_Counter>3){
344         Time_Interval_Counter=0;
345     }
346     if(Time_Interval_Counter==0){
347         Time_Interval_of_1ms_Elapsed=TRUE;
348         Time_Interval_Tick_Tock=~Time_Interval_Tick_Tock;
349     }
350     if(Time_Interval_Counter==2){

```



```

351         Time_Interval_of_500us_Elapsed=TRUE;
352     }
353     CLEAR_THE_TIMER_INTERRUPT_FLAG;
354     CLEAR_THE_PIE_INTERRUPT_FLAG_OF_THE_TIMER;
355 }

```

Il timer, impostato a $250\mu s$, ogni quattro conteggi modifica la seconda variabile in rosso come `TRUE`; ogni $1ms$ perciò si può entrare all'interno del `if` presente nel *main* (in caso di dubbi vedere *Control_Main.c* Code 1.1).

1.2.3 Algoritmo di controllo

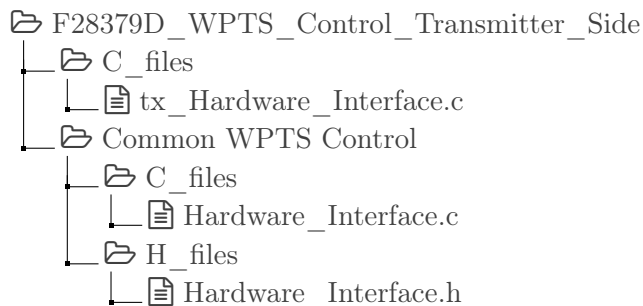
Menzioneremo solo alcuni aspetti presenti in `tx_Control_Algorithm_v1.c`, file nel quale si calcola di fatto l'Eq 1.1. Viene riportata una modalità di funzionamento particolare. Si vuole generare una tensione con ampiezza che varia in maniera sinusoidale e si applica una correzione rispetto ad un valore di offset. Quest'ultimo si ottiene quando le due onde del *phase shift* sono sfasate di $2\pi/3$ perché in questa condizione non c'è terza armonica. L'ampiezza di prima armonica della tensione è:

$$V_s = \frac{4}{\pi} \cdot V_{DC} \sin\left(\frac{2\pi}{3}\right) = \frac{4}{\pi} \cdot \frac{\sqrt{3}}{2} \cdot V_{DC} = 1.102657790843584 \cdot V_{DC} \quad (1.2)$$

È interessante infine il calcolo del $\sin(\theta)$ di un angolo generico. L'operazione è effettuata con la function `Sin_Interp(theta)` che compie di fatto un'interpolazione a partire da una serie di valori `const float Sin_Table [128+1]` richiamata dal file `tx_Define_Specific_Variables.c`

1.2.4 Hardware Interface

Esistono tre file denominati *Hardware_Interface*, due in formato `.c` ed uno in `.h`; si trovano nella directory



L'importanza di questi file è assoluta, in quanto essi contengono la maggior parte delle informazioni sui registri inizializzati all'accensione del micro.

L'idea di fondo è quella di scrivere un codice che sia indipendente dai registri:

- *Hardware_Interface.c*: è il file più importante in quanto contiene la prima inizializzazione dell'hardware (vedere successiva Sezione 1.3) e altre funzioni rilevanti come `Initialize_the_Peripherals_Clock` e `Initialize_PWMX`, che come suggeriscono abilitano rispettivamente il clock e le configurazioni dei moduli pwm.
- *tx_Hardware_Interface.c*: le principali funzioni sono richiamate dal precedente file, `Initialize_the_Microprocessor_Peripherals` e per gli interrupt `Enable_the_Interrupts`, quando viene Inizializzato l'hardware.
- *Hardware_Interface.h*: utilizzando i `#define` è possibile sostituire i registri a dei nomi costanti. Inoltre, si definiscono dei numeri che sono caratteristici del MCU, come ad esempio `TIMER0_PERIOD_LO` per il timer e `HALF_OF_INVERTER_PWM_NOMINAL_PERIOD` per la pwm, che dipendono dal clock del sistema. Si capisce facilmente come quest'ultimo file sia sensibilmente diverso rispetto a quello del precedente microcontrollore.

1.3 Inizializzazione microcontrollore e delle periferiche

Vediamo la procedura di inizializzazione, che avviene cioè prima di abilitare l'algoritmo di controllo del sistema. Come sempre, le parti di codice riportate sono ridotte all'essenziale, per non distogliere l'attenzione dai passaggi basilari.

I paragrafi seguenti saranno tre:

1. inizializzazione hardware 1.3.1;
2. funzioni richiamate 1.3.2;
3. utilizzo `#define` 1.3.3;

i codici riportati fanno parte dei tre file descritti in **Hardware Interface 1.2.4**.

1.3.1 Inizializzazione hardware

La funzione che segue è stata già citata in *Control_Main.c* Code 1.1, dove era evidenziata in rosso. È di fondamentale importanza poiché richiama delle funzioni che agiscono direttamente sui registri di sistema.

All'interno di `Project▶Common WPTS Control▶C_files` si trova il file *Hardware_Interface.c* che contiene Code 1.4.

Code 1.4. *Hardware_Interface.c*

```

41 int Initialize_the_Hardware()
42 {
43     int Result=TRUE;
44
45     InitSysCtrl();    //include DisableDog, InitPeripheralClocks
46
47     DINT;             //Clear all interrupts and initialize PIE vector table
48
49     InitPieCtrl();
50     //Initialize the PIE control registers to their default state.
51
52     IER = 0x0000;
53     IFR = 0x0000;
54
55     InitPieVectTable();
56     Result&=Initialize_the_Microprocessor_Peripherals();
57     Result&=Initialize_the_Peripherals_Clock();
58     Result&=Enable_the_Interrupts();
59     Result&=Start_the_Timer();
60
61     return Result;
62 }
```

Le prime tre funzioni sono di sistema e presenti nelle librerie; in breve:

`InitSysCtrl` in ordine, disabilita il timer del watchdog, eventualmente abilita la flash ed infine imposta il PLL a 200MHz;

`InitPieCtrl` inizializza i registri di controllo del PIE, disabilitando gli interrupt (PIEIER) e pulendo i flag (PIEIFR);

`InitPieVectTable` inizializza la tabella dei vettori del PIE.

Le successive funzioni, evidenziate in rosso, sono descritte nel paragrafo seguente.

1.3.2 Funzioni richiamate

Le *funzioni richiamate* servono per le configurazioni iniziali; verranno esposte solo alcune righe di codice delle diverse centinaia presenti. Si tratta di:

`Initialize_the_Microprocessor_Peripherals` vengono configurate le periferiche usando i registri di sistema in Code 1.5;

`Initialize_the_Peripherals_Clock` come suggerisce il nome, abilita il clock ai moduli utilizzati e lo disabilita agli altri, vedere Code 1.6;

`Enable_the_Interrupts` abilita i diversi interrupt e imposta la tabella dei vettori del PIE, Code 1.7;

sono quindi riportate in sintesi nei Code a seguire.

Code 1.5. tx_Hardware_Interface.c

```

39 int Initialize_the_Microprocessor_Peripherals()
40 {
41     int Result=TRUE;
42     /*      INITIALIZE SCI-A      */
43     ...
44     /*      INITIALIZE SCI-B      */
45     ...
46     /*      INITIALIZE SPI-A      */
47     ...
48     /*      INITIALIZE ADC        */
49     ...
50     CalAdcINL(ADC_ADCA);
51     DELAY_US(5000L);    // Waits for 5ms
52     /*      INITIALIZE PWM        */
53     EALLOW;
54     CpuSysRegs.PCLKCR0.bit.TBCLKSYNC=0;
55     ...
56     Initialize_PWMX(&EPwm1Regs, PHASE_SHIFT, HALF_OF..., LAG...);
57     Initialize_PWMX(&EPwm2Regs, PHASE_SHIFT, HALF_OF..., LEAD...);
58     Initialize_PWMX(&EPwm3Regs, OFF, 0, 0);
59     Initialize_PWMX(&EPwm4Regs, ADC_SOC, HALF_OF..., 0);
60
61     /*      SETUP of pins used as TZx      */
62     ...
63     CpuSysRegs.PCLKCR0.bit.TBCLKSYNC=1;
64     EDIS;
65
66     /*      SETUP of Timer 0      */
67     EALLOW;
68     CpuTimer0.RegAddr = &CpuTimer0Regs;
69     //CpuTimer0Regs.TCR.bit.TSS=1; // Stops Timer0
70     CpuTimer0Regs.TCR.bit.FREE=0;
71     CpuTimer0Regs.TCR.bit.SOFT=1;
72     CpuTimer0Regs.TCR.bit.TIF=1;    // Clears interrupt flag, if any
73     CpuTimer0Regs.TCR.bit.TIE=1;    // Timer0 interrupt enabled
74     // Divides the SYSCLKOUT by (TIMER0_CLOCK_DIVIDER+1)
75     CpuTimer0Regs.TPR.bit.TDDR=TIMER0_CLOCK_DIVIDER_LO;
76     CpuTimer0Regs.TPRH.bit.TDDR=TIMER0_CLOCK_DIVIDER_HI;
77     CpuTimer0Regs.PRD.bit.LSW=TIMER0_PERIOD_LO;
78     CpuTimer0Regs.PRD.bit.MSW=TIMER0_PERIOD_HI;
79     CpuTimer0Regs.TCR.bit.TRB=1;    //Initializes counter & prescaler
80     EDIS;
81
82     /*      SETUP GPIO20-23 and GPIO32,34 as digital I/O      */
83     ...
84
85     /*      SETUP of Chip Select pin for      */
86     /*      MCP23S17, 25LC160B, and nRF24L01+      */
87     ...

```

```

/*      SETUP of Chip Enable pin for      */
/*      nRF24L01+                          */
...

/*      SETUP of IRQ input pin from        */
/*      nRF24L01+                          */
...
672 return Result;
673 }

```

Per capire il significato dei registri bisogna far uso del manuale. In questa parte del programma si possono capire le configurazioni eseguite e a quali GPIO corrispondono i diversi moduli.

L'unica nota colorata (di rosso) si ha per `Initialize_PWMX` che imposta i moduli pwm; non è trattata di seguito, ma si può trovare in *Hardware_Interface.c*.

Il *clock* delle periferiche invece si abilita come segue.

Code 1.6. Hardware_Interface.c

```

112 int Initialize_the_Peripherals_Clock(void){
113     int Result=TRUE;
114     EALLOW;
115     //LSPCLK= SYSCLK/4 (default) =50MHz
116     ClkCfgRegs.LOSPCP.bit.LSPCLKDIV=2;
117     //EPWMCLK= PLLSYSCLK/2 (default) =100MHz
118     ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV=1;
119
120     // Enables the clock for the used peripherals
121     CpuSysRegs.PCLKCR0.bit.HRPWM = 1;    //enable HRPWM clock
122     CpuSysRegs.PCLKCR0.bit.TBCLKSYNC=0;  //Time Base Clock sync
123
124     CpuSysRegs.PCLKCR13.bit.ADC_A = 1;    // ADC
125     CpuSysRegs.PCLKCR13.bit.ADC_B = 0;
126     CpuSysRegs.PCLKCR13.bit.ADC_C = 0;
127     CpuSysRegs.PCLKCR13.bit.ADC_D = 0;
128     ...
129     CpuSysRegs.PCLKCR7.bit.SCI_A = 1;     // SCI-A
130     CpuSysRegs.PCLKCR7.bit.SCI_B = 1;     // SCI-B
131     CpuSysRegs.PCLKCR7.bit.SCI_C = 0;
132     CpuSysRegs.PCLKCR7.bit.SCI_D = 0;
133
134     CpuSysRegs.PCLKCR8.bit.SPI_A = 1;     // SCI-A
135     CpuSysRegs.PCLKCR8.bit.SPI_B = 0;     // SPI-B
136     CpuSysRegs.PCLKCR8.bit.SPI_C = 0;
137     ...
138     CpuSysRegs.PCLKCR2.bit.EPWM1 = 1;     // ePWM1
139     CpuSysRegs.PCLKCR2.bit.EPWM2 = 1;     // ePWM2
140     CpuSysRegs.PCLKCR2.bit.EPWM3 = 1;     // ePWM3
141     CpuSysRegs.PCLKCR2.bit.EPWM4 = 1;     // ePWM4
142     CpuSysRegs.PCLKCR2.bit.EPWM5 = 0;     // ePWM5
143     CpuSysRegs.PCLKCR2.bit.EPWM6 = 0;     // ePWM6
144     CpuSysRegs.PCLKCR2.bit.EPWM7 = 0;     // ePWM6
145     CpuSysRegs.PCLKCR2.bit.EPWM8 = 0;     // ePWM8 ... ePWM12
146     ...
147     CpuSysRegs.PCLKCR0.bit.CPUTIMER0 = 1; //enable Timer 0 clock
148     CpuSysRegs.PCLKCR0.bit.CPUTIMER1 = 0; //disable Timer 1 clock
149     CpuSysRegs.PCLKCR0.bit.CPUTIMER2 = 0; //disable Timer 2 clock

```

```

150 ...
151 CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1; //enables the clock
152 EDIS;
153 return Result;
154 }

```

Risulta abbastanza lineare, l'unica accortezza è quella di porre TBCLKSYNC=0 all'inizio per sincronizzare i moduli.

L'ultima funzione `Enable_the_Interrupts` è divisa in tre parti: la prima inserisce l'indirizzo della isr; la seconda abilita l'interrupt; la terza infine applica un *enable* a livello della CPU.

Code 1.7. tx_Hardware_Interface.c

```

683 int Enable_the_Interrupts(void)
684 {
685     int Result=TRUE;
686     EALLOW;
687     PieVectTable.TIMER0_INT = &CPU_Timer0_isr;
688     PieVectTable.SCIA_RX_INT = &SCIA_Rx_isr;
689     PieVectTable.SCIB_RX_INT = &SCIB_Rx_WPTS_isr;
690     PieVectTable.XINT1_INT=&RF_transceiver_isr;
691     PieVectTable.EPWM1_TZ_INT = &PWM_TZ1_isr;
692     PieVectTable.EPWM2_TZ_INT=&PWM_TZ1_isr;
693     PieVectTable.EPWM3_TZ_INT=&PWM_TZ1_isr;
694     PieVectTable.ADCA1_INT = &ADC_isr;
695     EDIS;
696     // Globally disable the interrupts
697     DINT;
698     EDIS;
699     // Disables the interrupts and clear any interrupt flag
700     IER = 0x0000;
701     IFR = 0x0000;
702
703     // Enables the used interrupts
704     PieCtrlRegs.PIEIER1.bit.INTx7=1;    // Timer0
705     PieCtrlRegs.PIEIER9.bit.INTx1=1;    // SCI-A Rx
706     PieCtrlRegs.PIEIER9.bit.INTx3=1;    // SCI-B Rx
707     PieCtrlRegs.PIEIER1.bit.INTx4=1;    // RF transceiver
708     PieCtrlRegs.PIEIER2.bit.INTx1=1;    // Trip Zone PWM1
709     PieCtrlRegs.PIEIER2.bit.INTx2=1;    // Trip Zone PWM2
710     PieCtrlRegs.PIEIER2.bit.INTx3=1;    // Trip Zone PWM3
711     PieCtrlRegs.PIEIER1.bit.INTx1=1;    // ADCA1
712     // Enables the interrupts at the CPU level
713     IER = M_INT1;    // ADC, for Timer0, and the RF transceiver
714     IER = M_INT2;    // Trip Zone
715     IER = M_INT9;    // SCI-A, SCI-B
716
717     EINT;    // Globally enables the interrupts
718     ERTM;    // Enables the debugger interrupt
719
720     return Result;
721 }

```

1.3.3 Direttiva #define

In conclusione alla parte di inizializzazione del micro, voglio citare un ulteriore frammento del codice che compone il programma, per dare l'idea di come siano strutturati i `#define`.

All'interno di Code 1.8 una prima porzione assegna ai registri dei nomi facili da ricordare; al momento della compilazione i nomi sono sostituiti con le definizioni originali. Una seconda invece, tratta numeri destinati al funzionamento di pwm e timer.

Code 1.8. Hardware_Interface.h

```

171 #define DUTY_CYCLE_A_PWM1          EPwm1Regs.CMPA.bit.CMPA
172 #define DUTY_CYCLE_A_PWM1_HR       EPwm1Regs.CMPA.bit.CMPAHR
173 #define DUTY_CYCLE_B_PWM1          EPwm1Regs.CMPB.bit.CMPB
174 #define DUTY_CYCLE_B_PWM1_HR       EPwm1Regs.CMPB.bit.CMPBHR
175 #define DUTY_CYCLE_A_PWM2          EPwm2Regs.CMPA.bit.CMPA
176 #define DUTY_CYCLE_A_PWM2_HR       EPwm2Regs.CMPA.bit.CMPAHR
177 #define DUTY_CYCLE_B_PWM2          EPwm2Regs.CMPB.bit.CMPB
178 #define DUTY_CYCLE_B_PWM2_HR       EPwm2Regs.CMPB.bit.CMPBHR
179 #define DUTY_CYCLE_A_PWM3          EPwm3Regs.CMPA.bit.CMPA
180 #define DUTY_CYCLE_B_PWM3          EPwm3Regs.CMPB.bit.CMPB
181 #define DUTY_CYCLE_A_PWM4          EPwm4Regs.CMPA.bit.CMPA
182 #define DUTY_CYCLE_B_PWM4          EPwm4Regs.CMPB.bit.CMPB

195 //SYSCLKOUT=200 MHz, ePWMCLK=100 MHz
196 //PWM period = 11.76 us, PWM frequency = 85.034 kHz
197 #define HALF_OF_INVERTER_PWM_NOMINAL_PERIOD 588
198 #define QUARTER_OF_INVERTER_PWM_NOMINAL_PERIOD 294
199 //PWM period = 12.30 us, PWM frequency = 81.301 kHz
200 #define HALF_OF_INVERTER_PWM_MAXIMUM_PERIOD 615
201 #define QUARTER_OF_INVERTER_PWM_MAXIMUM_PERIOD 307.5
202 //PWM period = 11.10 us, PWM frequency = 90.090 kHz
203 #define HALF_OF_INVERTER_PWM_MINIMUM_PERIOD 555
204 #define QUARTER_OF_INVERTER_PWM_MINIMUM_PERIOD 277.5
205 //PWM period = 46.97 us, PWM frequency = 21.258 kHz
206 #define HALF_OF_CHOPPER_NOMINAL_PERIOD 2352
207 #define QUARTER_OF_CHOPPER_NOMINAL_PERIOD 1176
208 //Dead Band = 0.53 us
209 #define DEAD_TIME 53
210 #define TIME_BASE_4_PHASE_REGISTER 0
211
212 // PWM Methods
213 #define OFF 0
214 #define CHOPPER 1
215 #define PHASE_SHIFT 2
216 #define DOM 3
217 #define ADC_SOC 4
218 #define LEADING_PHASE 1
219 #define LAGGING_PHASE 2

237 /* Management of the timer */
238 // Timer0 period is 250us, SYSCLKOUT = 200 MHz
239 #define TIMER0_PERIOD_LO 250
240 #define TIMER0_PERIOD_HI 0
241 #define TIMER0_CLOCK_DIVIDER_LO 199
242 #define TIMER0_CLOCK_DIVIDER_HI 0
243 #define START_THE_TIMER CpuTimer0Regs.TCR.bit.TSS=1

```

```

244 | #define CLEAR...INTERRUPT_FLAG      CpuTimer0Regs.TCR.bit.TIF=1
245 | #define CLEAR...FLAG_OF_THE_TIMER  PieCtrlRegs.PIEACK.all=PIEACK_GROUP1

```

I numeri sopra evidenziati dipendono dal funzionamento dei moduli pwm e del timer. In sintesi essi si ricavano:

$$\text{HALF_OF_INVERTER_PWM} \dots \text{PERIOD} = \frac{\text{ePWCLOCK}}{f_{PWM} \cdot 2} = \frac{100MHz}{85\,034Hz \cdot 2} = 588 \quad (1.3)$$

$$\frac{\text{TIMER0_CLOCK_DIVIDER} + 1}{\text{SYSCLKOUT}} \cdot \text{TIMER0_PERIOD} = \frac{199 + 1}{200MHz} \cdot 250 = 250\mu s \quad (1.4)$$

Dopo aver visto gli oggetti presenti sul banco di lavoro, capito la struttura del programma ed infine compreso i passaggi eseguiti per l'inizializzazione, risulta ovvio voler approfondire i moduli citati e cercare delle possibili migliorie al sistema. È questa la funzione del Capitolo 2.

Capitolo 2

Moduli utilizzati

Nel presente capitolo non si vedranno tutti i moduli utilizzati dal software, ma solamente i più importanti o quelli modificati al fine di funzionare correttamente con il TMS320F28379D. Altre componenti, se non menzionate, risultano poco significative o non presentano correzioni nel codice rispetto alla versione del precedente microcontrollore.

2.1 Clocking

Il clock del dispositivo raggiunge i diversi moduli, ed una presentazione generale si può vedere in Fig 2.1 tratta dal paragrafo *3.7 Clocking, Technical Reference Manual* [1]; ci sono quattro possibili sorgenti di clock:

INTOSC2 *Oscillatore interno primario*, la frequenza 10MHz, è il clock di sistema di default;

INTOSC1 *Oscillatore interno secondario*, la frequenza 10MHz, è una sorgente di backup e di ridondanza;

XTAL *Oscillatore esterno*, sono presenti due pin X1 e X2 per una sorgente di clock esterna; nel caso in cui tra i *predefined symbols* sia presente `_LAUNCHXL_F28379D` è questa la fonte di clock utilizzata, ed è pari a 10MHz;

AUXCLKIN *Input clock ausiliario*, è supportato un clock esterno addizionale sul GPIO133; può essere usato come sorgente per USB e CAN.

Tra le *derivazioni di clock* le più importanti risultano:

- **PLLSYSCLK** arriva a GS RAM, modulo IPC, qualificazione GPIO e NMI watchdog timer; l'impostazione di tale frequenza è ampiamente trattata in seguito e si conclude con la sua determinazione nell'Eq 2.1;

trollers datasheet [5]); il registro corrispondente è lasciato di default `ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV=1` dividendo per due il `PLLSYSCLK`.

Per quanto riguarda l'inizializzazione del clock, già trattata in Code 1.4 (`function Initialize_the_Hardware()`), risulta fondamentale la funzione `InitSysCtrl` (file *F2837xD_SysCtrl.c*) all'interno della quale è possibile trovare una delle configurazioni del `PLLSYSCLK`.

Code 2.1. F2837xD_SysCtrl.c

```

94 void InitSysCtrl(void)
   {...
162 #ifdef _LAUNCHXL_F28379D
163   InitSysPll(XTAL_OSC, IMULT_40, FMULT_0, PLLCLK_BY_2);
164 #else
165   InitSysPll(XTAL_OSC, IMULT_20, FMULT_0, PLLCLK_BY_2);
166 #endif // _LAUNCHXL_F28379D
167 ...}

```

L'impostazione che avviene in `InitSysPll(...)` dipende dalle costanti definite in *F2837xD_Examples.h* e fanno riferimento, in ordine, ai registri:

`ClkCfgRegs.CLKSRCTL1.bit.OSCCLKSRCSEL` per il `LAUNCHXL_F28379D` la sorgente di clock è imposta `XTAL`;

`ClkCfgRegs.SYSPLLMULT.bit.IMULT` parte intera del moltiplicatore;

`ClkCfgRegs.SYSPLLMULT.bit.FMULT` parte frazionaria del moltiplicatore;

`ClkCfgRegs.SYSCLKDIVSEL.bit.PLLSYSCLKDIV` divisore del clock di sistema;

e si compongono in modo da determinare la frequenza:

$$f_{\text{PLLSYSCLK}} = f_{\text{OSCCLK}} \cdot \frac{\text{IMULT} + \text{FMULT}}{\text{SYSCLKDIVSEL}} = 10\text{MHz} \cdot \frac{40 + 0}{2} = 200\text{MHz} \quad (2.1)$$

che risulta la massima per questo microcontrollore.

2.2 Timer

Abbiamo già illustrato lo scopo del timer, quello cioè di produrre ogni $250\mu\text{s}$ un segnale di interrupt, ma è interessante capire come i registri imposti in Code 1.5 (`function Initialize_the_Microprocessor_Peripherals()`) interagiscano con il `SYSCLK`.

Sono tre i `TIMER(0/1/2)` a 32 bit presenti; il funzionamento si palesa in Fig 2.2a dove i registri menzionati:

TIMH:TIM *counter*, il registro a 32 bit viene caricato con il valore PRDH:PRD e decrementato ogni volta che PSCH:PSC arriva a zero; quando raggiunge lo zero viene generato un impulso di interrupt (per le impostazioni eseguite ogni $250\mu s$).

PRDH:PRD *timer period*, è il valore caricato ad ogni ciclo all'interno di TIMH:TIM; nel codice è posto `CpuTimer0Regs.PRD.bit.LSW=250`.

TDDRH:TDDR *timer divide-down*, è il numero caricato in PSCH:PSC ad ogni ciclo; come impostazione `CpuTimer0Regs.TPR.bit.TDDR=199`

PSCH:PSC *prescale counter*, partendo dal caricamento del precedente registro, ad ogni impulso di SYSCLK il valore viene decrementato; lo zero si ottiene ogni $(TPR[TDDRH:TDDR]+1)$.

È possibile dunque individuare un ciclo "veloce" nella parte sinistra dello schema 2.2a, dove *prescale counter* conta a 200MHz, e un ciclo "lento" nella parte destra con decremento ogni $1\mu s$.

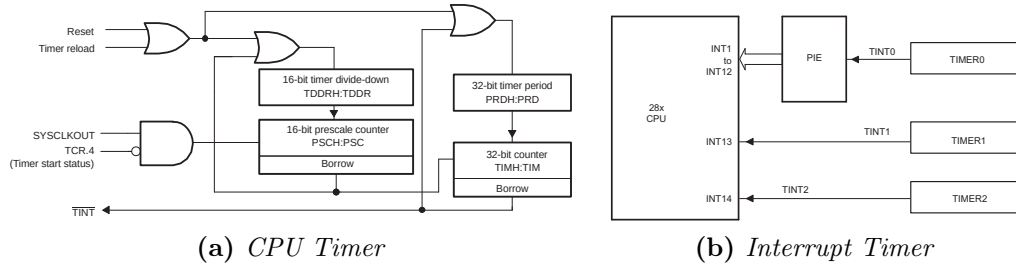


Figura 2.2. Struttura Timer

I segnali di interrupt TINT0, TINT1 e TINT2 sono connessi come in Fig 2.2b; da notare che solo il segnale del TIMER0 passa attraverso il PIE.

Infine si verifica il corretto funzionamento della periferica attraverso la visualizzazione di segnali con l'oscilloscopio: per mezzo dei TOGGLE è possibile controllare l'operatività del timer. All'interno dell'interrupt corrispondente è inserita questa funzione che permette di cambiare lo stato del pin preso in esame (in questo caso il pin test GPIO32). In Fig 2.3 è possibile notare come lo stato alto o basso permanga per $250\mu s$ come desiderato.

2.3 Peripheral Interrupts

Nel capitolo precedente abbiamo trattato le *interrupt service routine*; in questo paragrafo è utile approfondire la struttura di questo modulo.

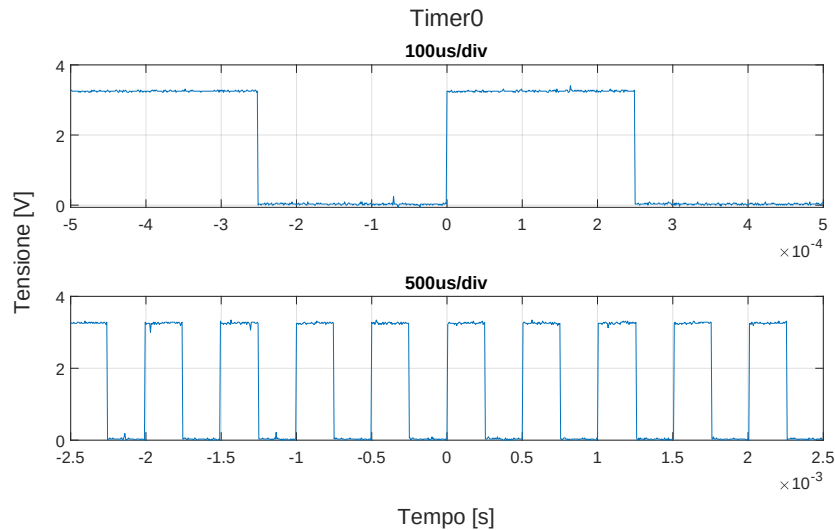


Figura 2.3. Interrupt Timer0, TOGGLE GPIO32 ogni 250 μ s

La CPU C28x ha quattordici linee di *peripheral interrupt*; due di queste, INT13 e INT14 sono connesse direttamente a TIMER1 e TIMER2 rispettivamente. I rimanenti dodici sono connessi attraverso modulo *Peripheral Interrupt Expansion*. Ad ogni linea interrupt è possibile collegare fino a sedici peripheral interrupt attraverso dei multiplexer. Si può vedere l'architettura completa in Fig 2.4.

Il percorso degli interrupt si può dividere in tre *stage*:

- *Peripheral* ogni periferica ha la propria configurazione unica di interrupt; possono esserci più eventi collegati allo stesso interrupt.
- *PIE* fornisce registri enable PIEIERx e flag PIEIFRx a 16 bit; un bit di acknowledge PIEACK fornisce una maschera all'intero gruppo PIE.
- *CPU* ha i propri registri enable IER, flag IFR e una maschera globale INTM.

Entrambe le CPU hanno il proprio PIE, che possono essere configurate indipendentemente.

La *sequenza di accesso* degli interrupt è illustrata in Fig 2.5. Quando una periferica genera un interrupt (gruppo Pie x, canale y) attiva la seguente catena di eventi:

1. l'interrupt è legato PIEIFRx.y;

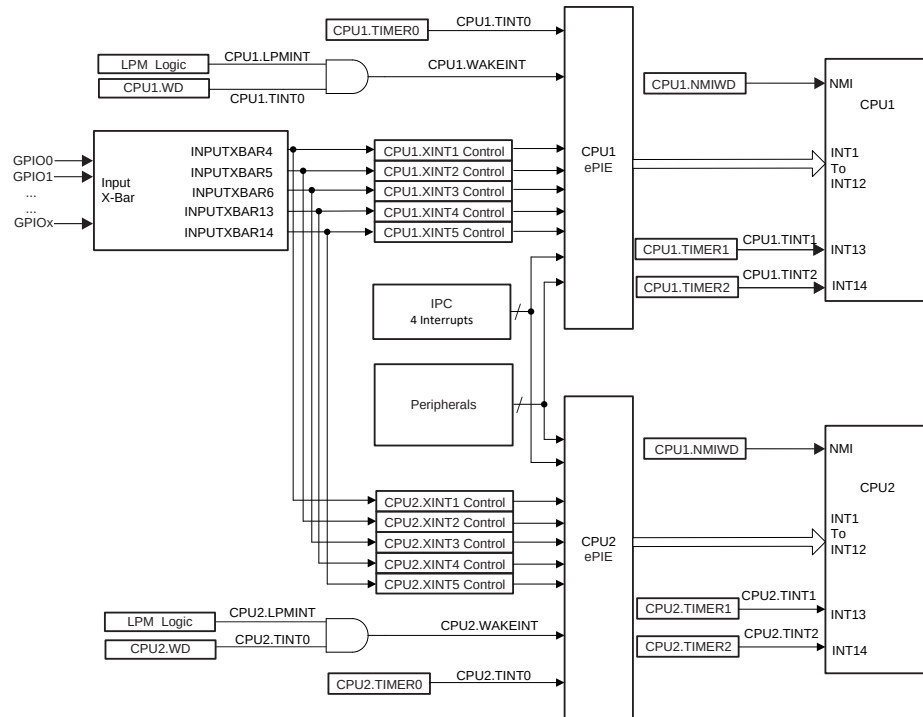


Figura 2.4. Architettura interrupt

2. se $PIEIERx.y$ è set, l'interrupt prosegue;
3. se $PIEACK.x$ è clear, l'interrupt prosegue e $PIEACK.x$ è set;
4. l'interrupt è legato $IFR.x$;
5. se $IER.x$ è set, l'interrupt prosegue;
6. se $INTM$ è clear, la CPU riceve l'interrupt;
7. la CPU salva il contesto;
8. $IFR.x$ e $IER.x$ sono cancellati, $INTM$ è set;
9. viene recuperata la *isr* dal PIE, $PIEFRx.y$ è clear

La *latenza dell'interrupt* è il tempo tra il primo passaggio e la prima istruzione *isr* eseguita dalla CPU; la latenza minima è di 14 cicli $SYSCLK$.

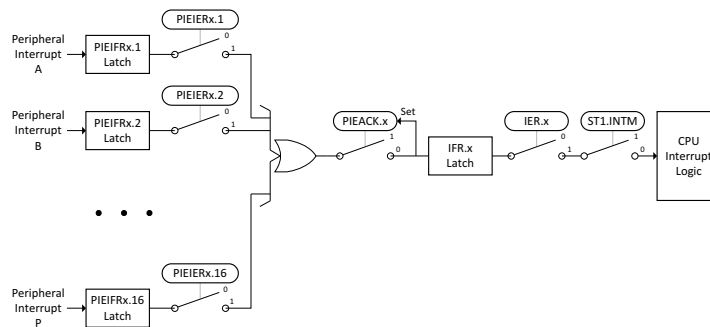


Figura 2.5. Percorso propagazione interrupt

2.4 GPIO

In un singolo pin abilitato GPIO possono arrivare fino a dodici segnali periferici indipendenti; sono presenti *sei porte* per gestire i 168 GPIO:

- **A** composta da GPIO0-31;
- **B** composta da GPIO32-63;
- **C** composta da GPIO64-95;
- **D** composta da GPIO96-127;
- **E** composta da GPIO128-159;
- **F** composta da GPIO160-168;

Non tutti i GPIO sono disponibili con i relativi pin nel *TI LaunchPad kit F28379D*; quelli presenti si possono individuare nel retro della scheda o nella serigrafia in Fig 1.2b.

Lo schema logico per un singolo pin si può vedere in Fig 2.6.

2.4.1 Registri di controllo

Per configurare e controllare i GPIO sono utilizzati due registri principali. Il primo `GpioCtrlRegs` si suddivide in:

MUX è possibile scegliere il *muxing* per i segnali delle periferiche visionando il paragrafo 8.7 *GPIO and Peripheral Muxing, Technical Reference Manual* [1]; di seguito, al Paragrafo 2.4.2, è presente un approfondimento sui due registri coinvolti:

```
GpioCtrlRegs.GPyGMUXn.bit.GPIOz
GpioCtrlRegs.GPyMUXn.bit.GPIOz
```

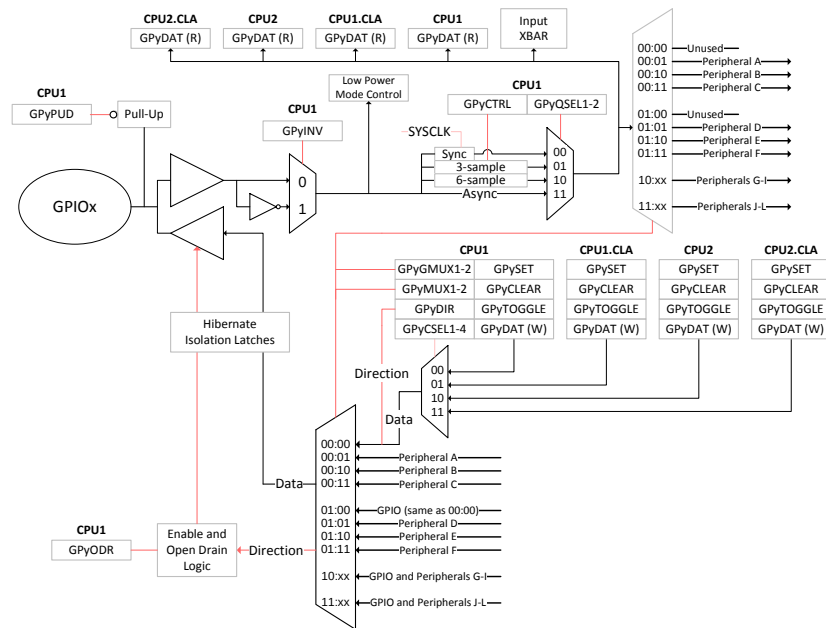


Figura 2.6. GPIO Logica di un singolo pin

PUD il *pullup* può essere usato per mantenere l'input pin in uno stato noto quando non è pilotato da un segnale esterno; tutti i pullup sono disabilitati di default e il registro è `GpioCtrlRegs.GPyPUD.bit.GPIOz`

QUAL se il pin è usato come input, la *qualificazione* del segnale è selezionabile attraverso `GpioCtrlRegs.GPyQSELn.bit.GPIOz`; di default tutte le qualificazioni sono sincrone con il PLLSYSCLK; per la spiegazione della qualificazione, vedere Paragrafo 2.4.3

CTRL il *periodo di campionamento* della qualificazione dell'ingresso è impostato tramite `GpioCtrlRegs.GPyCTRL.bit.QUALPRDn=33`; nell'esempio il clock è di 200MHz, il periodo di sampling risulta perciò $\frac{\text{PLLSYSCLK}}{33 \cdot 2} = 3.03\text{MHz}$;

DIR seleziona la *direzione* del pin come input o come output; l'istruzione `GpioCtrlRegs.GPyDIR.bit.GPIOz` si pone =0 per input (default), =1 per output

il secondo dei due registri principali `GpioDataRegs` contiene i seguenti comandi che coinvolgono lo stato del pin GPIO:

DAT la lettura del `GpioDataRegs.GPyDAT.bit.GPIOz` riflette lo stato corrente del pin; la scrittura imporrà il valore alto o basso se il pin è abilitato per l'output

SET imposta il pin con valore alto se `GpioDataRegs.GPySET.bit.GPIOz=1`

CLEAR cancella il valore del pin se `GpioDataRegs.GPyCLEAR.bit.GPIOz=1`

TOGGLE commuta il valore del pin se `GpioDataRegs.GPyTOGGLE.bit.GPIOz=1`

2.4.2 Muxing periferiche

Il microcontrollore F28379D incorpora un doppio livello di muxing: un multiplexer di gruppo e quattro GPIO Index multiplexer che permettono fino a dodici segnali periferici indipendenti. Lo schema è riportato in Fig 2.7.

Nella Tab 2.1 si possono vedere i GPIO utilizzati e il relativo muxing.

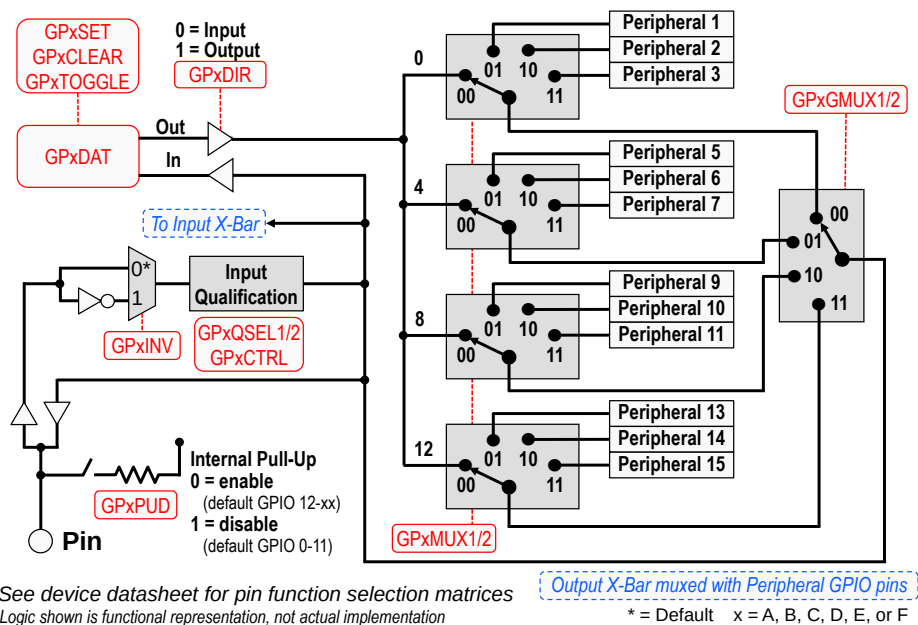


Figura 2.7. GPIO Diagramma blocchi pin

Le configurazioni già accennate nell'**Inizializzazione hardware 1.3.1** vedevano nel file `tx_Hardware_Interface.c` (Code 1.5) l'utilizzo di registri del microcontrollore; tra questi sono presenti anche la gestione dei GPIO. Riporto di seguito un estratto dallo stesso file che risulterà familiare a chi, prestando attenzione, ha letto la `ADC_isr` (Code 1.2): la `function Enable_PWM()` ha lo scopo di impostare il mux corretto per i moduli pwm utilizzati.

Code 2.2. `tx_Hardware_Interface.c`

```

764 int Enable_PWM(void){
765     int Result=TRUE;
766     EALLOW;

```

```

767 GpioCtrlRegs.GPAMUX1.bit.GPIO0 = 1; //EPWM1A
768 GpioCtrlRegs.GPAMUX1.bit.GPIO1 = 1; //EPWM1B
769 GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 1; //EPWM2A
770 GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 1; //EPWM2B
771 GpioCtrlRegs.GPAMUX1.bit.GPIO4 = 1; //EPWM3A
772 GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 1; //EPWM3B
773 GpioCtrlRegs.GPAMUX1.bit.GPIO6 = 1; //EPWM4A
774 GpioCtrlRegs.GPAMUX1.bit.GPIO7 = 1; //EPWM4B
775 EDIS;
776 return Result;
777 }

```

2.4.3 Qualificazione del segnale

Nel caso di un pin GPIO input, la qualificazione può essere specificata secondo le opzioni disponibili:

- *Asincrona* il segnale in ingresso non è sincronizzato con il clock di sistema;
- *Sincrona* il segnale viene sincronizzato con SYSCLKOUT, e questo può comportare un delay fino ad un periodo del clock;
- *Finestra di campionamento* il segnale viene sincronizzato con SYSCLKOUT e qualificato di un numero specifico di cicli pari a 3 o 6 sample; è possibile impostare il periodo sampling con il registro GPyCTRL. In Fig 2.8 oltre allo schema a blocchi, è presente un esempio con GPyQSELn=2 e GPyCTRL=1: si traduce in 6 sample di frequenza pari a SYSCLKOUT/2, la finestra di campionamento lunga $5 \cdot (T_{\text{SYSCLK}} \cdot 2)$ permette dunque di ignorare il glitch presente, ma introduce allo stesso tempo un ritardo sui fronti rispetto al segnale originale.

2.5 Crossbar (X-BAR)

Il modulo *crossbar* fornisce la possibilità di connettere in modo flessibile gli ingressi, le uscite e le risorse interne del dispositivo in molteplici configurazioni. In totale il device contiene quattro X-BAR: Input, Output, CLB ed ePWM X-BAR. Per l'uso del microcontrollore verrà esaminata sola la prima, relativa agli ingressi.

2.5.1 Input X-BAR

L'*Input X-BAR* viene utilizzato per instradare i segnali da un GPIO a molti blocchi IP diversi come ADC, eCAP, ePWM e gli interrupt esterni.

Tabella 2.1. GPIO Muxed Pin

GPIO	GPyGMUXn. GPIOz	GPyMUXn. GPIOz	Periferica
GPIO0	GPAGMUX1=0	GPAMUX1=1	ePWM1A
GPIO1	GPAGMUX1=0	GPAMUX1=1	ePWM1B
GPIO2	GPAGMUX1=0	GPAMUX1=1	ePWM2A
GPIO3	GPAGMUX1=0	GPAMUX1=1	ePWM2B
GPIO4	GPAGMUX1=0	GPAMUX1=1	ePWM3A
GPIO5	GPAGMUX1=0	GPAMUX1=1	ePWM3B
GPIO6	GPAGMUX1=0	GPAMUX1=1	ePWM4A
GPIO7	GPAGMUX1=0	GPAMUX1=1	ePWM4B
GPIO29	GPAGMUX2=0	GPAMUX2=1	SCITXDA
GPIO64	GPCGMUX1=1	GPCMUX1=2	SCIRXDA
GPIO9	GPAGMUX1=0	GPAMUX1=2	SCITXDB
GPIO11	GPAGMUX1=0	GPAMUX1=2	SCIRXDB
GPIO16	GPAGMUX2=0	GPAMUX2=1	SPISIMOA
GPIO18	GPAGMUX2=0	GPAMUX2=1	SPICLKA
GPIO59	GPBGMUX2=3	GPBMUX2=3	SPISOMIA
GPIO25	GPAGMUX2=0	GPAMUX2=0	INPUT1 (Xbar, TZ1)
GPIO27	GPAGMUX2=0	GPAMUX2=0	INPUT2 (Xbar, TZ2)
GPIO14	GPAGMUX1=0	GPAMUX1=0	INPUT3 (Xbar, TZ3)
GPIO24	GPAGMUX2=0	GPAMUX2=0	INPUT4 (Xbar, XINT1)
GPIO15	GPAGMUX1=0	GPAMUX1=0	INPUT6 (Xbar, TRIP6)
GPIO22	GPAGMUX2=0	GPAMUX2=0	TEST
GPIO32	GPBGMUX1=0	GPBMUX1=0	TEST
GPIO34	GPBGMUX1=0	GPBMUX1=0	LED TEST
GPIO8	GPAGMUX1=0	GPAMUX1=0	output (MCP23S17)
GPIO10	GPAGMUX1=0	GPAMUX1=0	output (25LC160B)
GPIO26	GPAGMUX2=0	GPAMUX2=0	output (nRF24L01+)
GPIO67	GPCGMUX1=0	GPCMUX1=0	output (nRF24L01+)

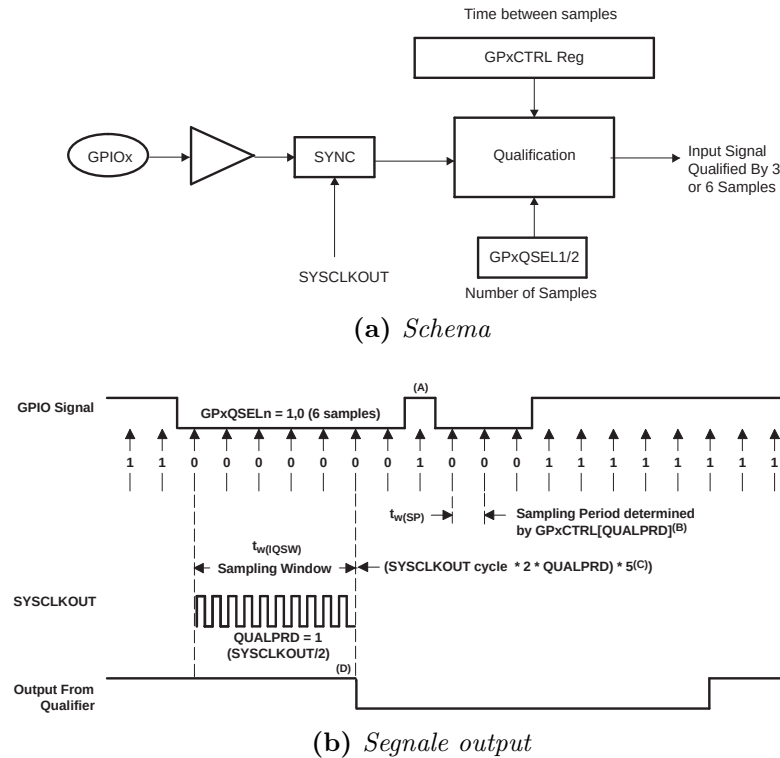


Figura 2.8. Qualificazione con finestra di campionamento

L'Input X-BAR è configurato tramite i registri `InputXbarRegs.INPUTnSELECT`; le destinazioni disponibili si vedono in Fig 2.9.

Le configurazioni degli input x-bar si trovano in `tx_Hardware_Interface.c` e nella `function Initialize_the_Microprocessor_Peripherals()`. Si possono vedere i registri coinvolti nella configurazione del microcontrollore in Tab 2.2: i *trip zone* si trovano alla fine della parte `Initialize PWM`, l'interrupt esterno è trattato nel paragrafo seguente.

2.5.1.1 External Interrupt

È doveroso aprire una piccola parentesi per quanto riguarda gli *external interrupt*: come riferisce il manuale, i pin `XINT1,2,3,4,5` devono essere impostati selezionando rispettivamente le sorgenti per i segnali `INPUT4,5,6,13,14`. Oltre a ciò, gli interrupt stessi devono essere abilitati ed avere impostata la polarità.

Code 2.3. tx_Hardware_Interface.c

```
39 | int Initialize_the_Microprocessor_Peripherals(){
40 | int Result=TRUE;
```

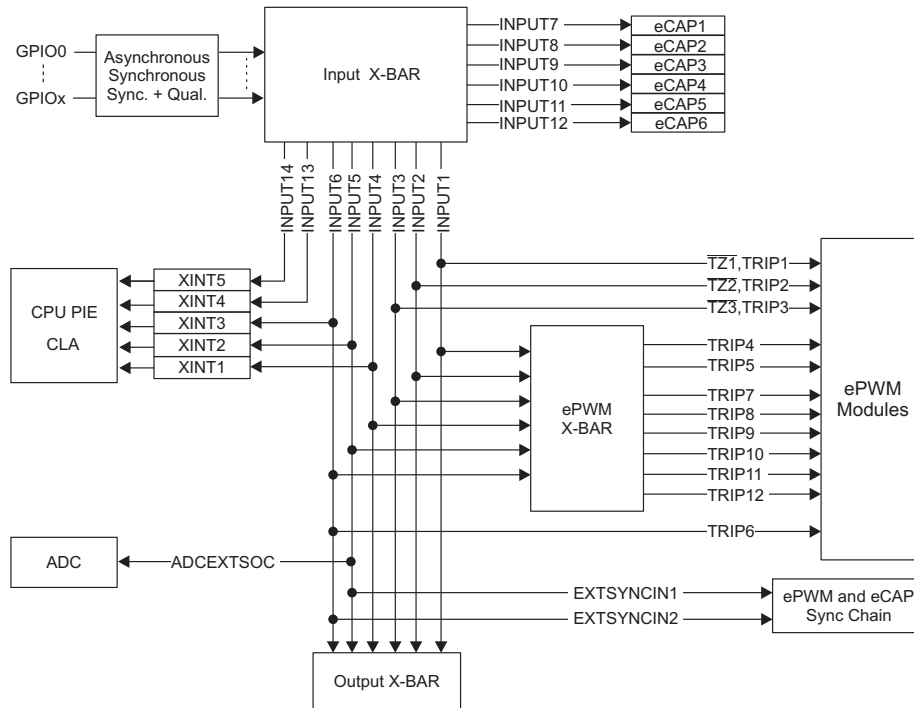


Figura 2.9. Input X-BAR

```

...
652 /*      Setup of IRQ input pin from nRF24L01+      */
653 EALLOW;
654 GpioCtrlRegs.GPAPUD.bit.GPIO24=0; //enables pull up
655 GpioCtrlRegs.GPADIR.bit.GPIO24=0; //sets pin as digital input
656 GpioCtrlRegs.GPAMUX2.bit.GPIO24=0; //the pin as digital i/o
657 //the input pin is sampled at 200MHz/(33*2)=3.03MHz
658 GpioCtrlRegs.GPACTRL.bit.QUALPRD3=33;
659 //qualification requires 3 samples -> 1us
660 GpioCtrlRegs.GPAQSEL2.bit.GPIO24=1;
661 InputXbarRegs.INPUT4SELECT=24; // select GPIO24
662 XintRegs.XINT1CR.bit.POLARITY=0; //XINT negative edge triggered
663 XintRegs.XINT1CR.bit.ENABLE=1; //XINT Enabled
664 EDIS;
665 return Result;
666 }

```

Nel Code 2.3 è riportata la configurazione del GPIO24 con `GpioCtrlRegs`, la selezione del `InputXbarRegs` ed infine l'abilitazione del relativo *interrupt esterno* con `XintRegs`.

Tabella 2.2. Destinazione input X-bar

InputXbarRegs. INPUTnSELECT=GPIOz	Destinazione
.INPUT1SELECT=25;	ePWM[TZ1,TRIP1]
.INPUT2SELECT=27;	ePWM[TZ2,TRIP2]
.INPUT3SELECT=14;	ePWM[TZ3,TRIP3]
.INPUT6SELECT=15;	ePWM[TRIP6]
.INPUT4SELECT=24;	XINT1

2.6 Enhanced Pulse Width Modulator

Il modulo della pwm è uno tra i più importanti del microcontrollore in quanto il suo compito è proprio quello di generare i segnali di comando per gli switch. All'interno del TMS320F28379D sono installati dodici moduli pwm.

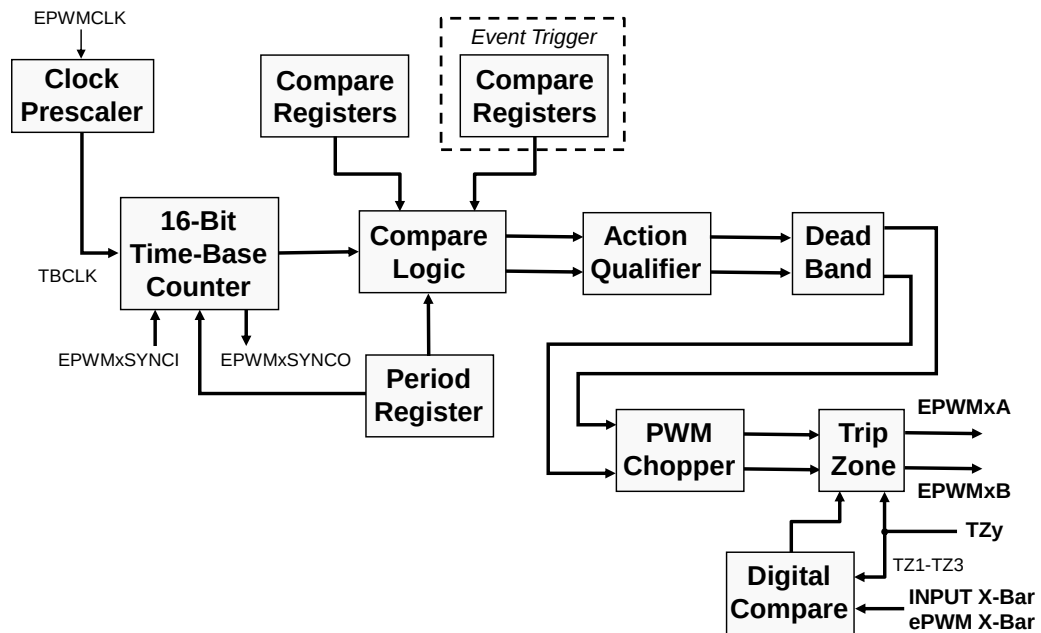


Figura 2.10. Schema a blocchi PWM

La sezione corrente presenterà in modo rapido tutte le componenti dei moduli, in modo da istruire il lettore nell'uso dei registri menzionati in **Configurazione ePWM 2.6.8**. I sottomoduli presenti sono:

- *Time Base* è collegato al EPWMCLK clock pwm, e prevede le configurazioni del contatore CTR, del modo di contare UP-DOWN e del periodo PRD;
- *Counter Compare* confronta il valore del TB counter con quattro registri CMPA-B-C-D e genera quattro eventi indipendenti che vengono inviati all'action qualifier;
- *Action Qualifier* è l'elemento chiave del modulo ePWM in grado di generare le forme d'onda PWM a partire dagli eventi in arrivo dal TB e CMP;
- *Dead-Band Generator* supporta ritardi del fronte di salita (RED) e di discesa (FED) programmabili indipendentemente con varie opzioni per la generazione delle uscite EPWMxA e EPWMxB;
- *PWM Chopper* modula con un segnale portante ad alta frequenza la forma d'onda PWM generata dal AQ e dal DB; in questo progetto non viene utilizzato;
- *Event Trigger* gestisce gli eventi generati dal TB, CMP e DC per generare un interrupt EPWMx_INT oppure impulsi di inizio conversione per l'ADC;
- *Trip Zone* agisce in caso di guasto con una logica indipendente e che porta le uscite EPWMxA e EPWMxB ad uno stato conosciuto (alto o basso);
- *Digital Compare* confronta i segnali esterni al modulo ePWM, ad esempio un segnale proveniente dai comparatori analogici CMPSS, per generare direttamente eventi o azioni PWM che vengono poi utilizzati dai sottomoduli trip-zone, time-base ed event-trigger.

I seguenti paragrafi tratteranno alcuni sottomoduli in modo più approfondito, e faranno riferimento alla Fig 2.10 per i segnali e registri.

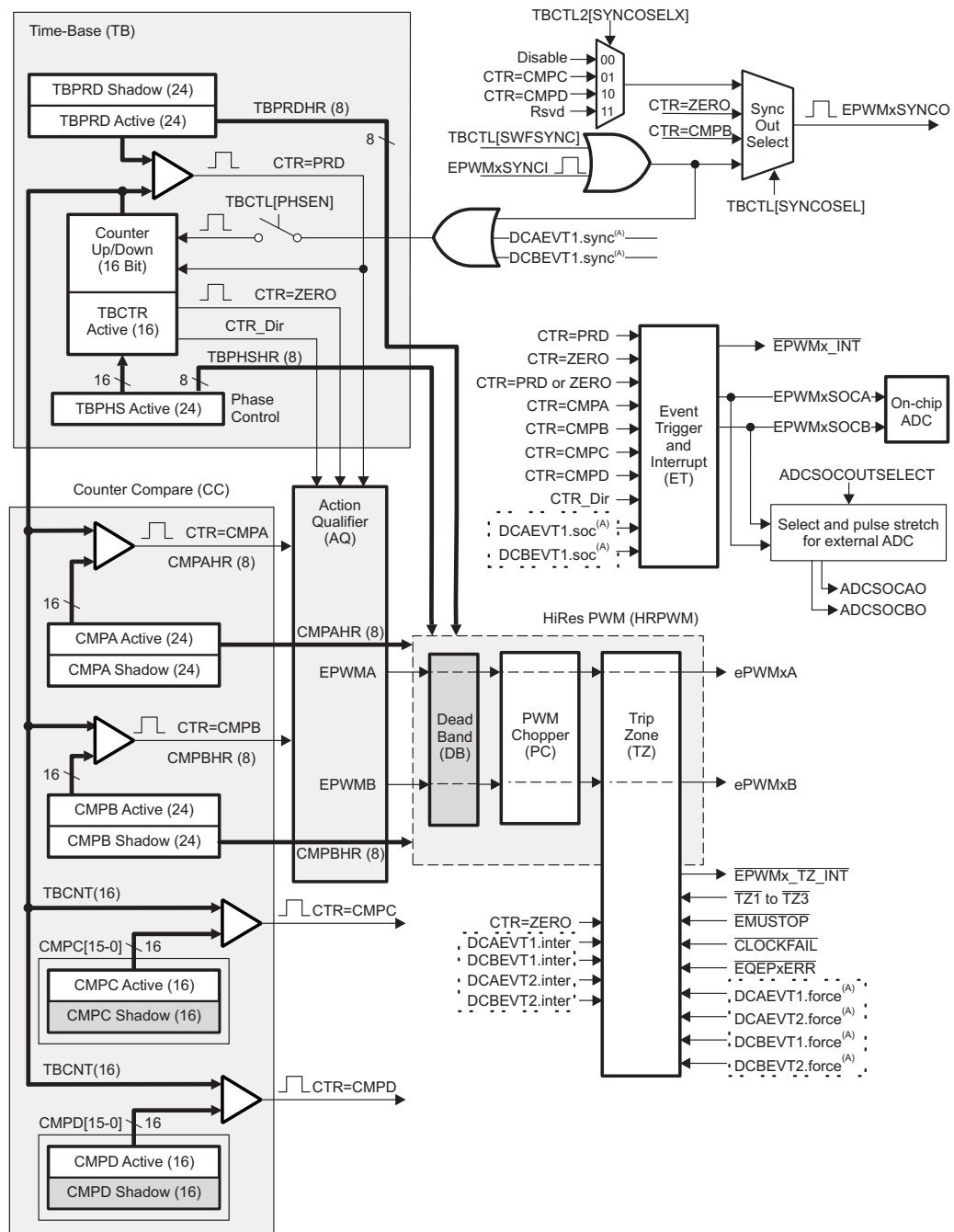


Figura 2.11. Moduli ePWM ed Interconnessione segnali interni

2.6.1 TB

Il *Time Base* è caratterizzato dai seguenti registri:

T_{BCLK} il *clock* del counter, e dipende da due registri che fungono da pre-scaler del clock pwm: $T_{BCLK} = EPWMCLK / (HSPCLKDIV * CLKDIV)$;

T_{BCTR} il *contatore* a 16 bit che viene incrementato o decrementato ad ogni impulso clock;

T_{BCTL} registro di *controllo*, tra le altre cose permette di impostare $CTRMODE$ la modalità di conteggio up, down e up-down;

T_{BPRD} definisce *periodo* del TB counter;

T_{BPHS} imposta *phase offset*, cioè la fase del TB counter.

Il sottomodulo inoltre genera due eventi, quando $CTR=0$ e $CTR=PRD$. Il periodo pwm si può calcolare in base al modo del counter; nel caso preso in esame, UP-DOWN, è descritto dalla seconda equazione:

$$T_{PWM} = (TBPRD+1) \cdot T_{TBCLK} \quad (2.2)$$

$$T_{PWM} = 2 \cdot TBPRD \cdot T_{TBCLK} = 2 \cdot 588 \cdot 10ns = 85034Hz \quad (2.3)$$

e si può vedere l'andamento temporale in Fig 2.12b.

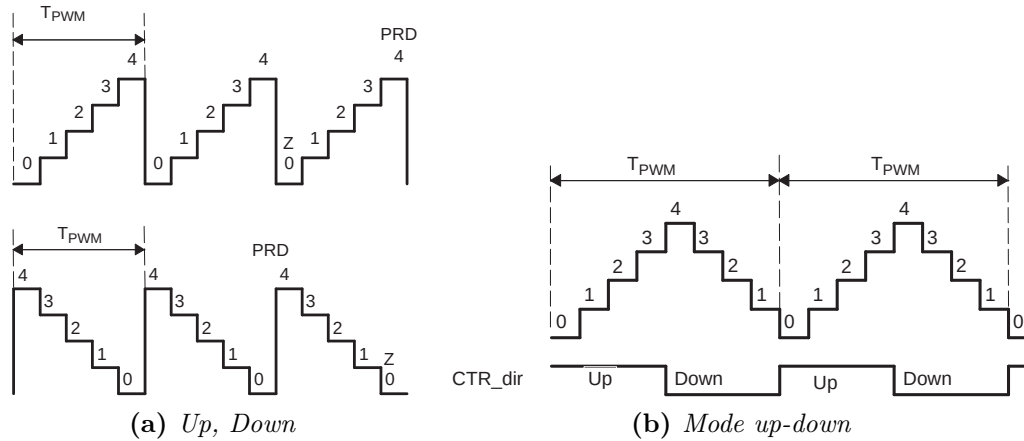


Figura 2.12. Counter mode

2.6.2 CMP

Nel *Counter Compare* tipicamente $CMPA$ e $CMPB$ vengono utilizzati per controllare il duty cycle della forma d'onda PWM generata; tutti e quattro i registri CMP possono essere utilizzati per avviare una conversione ADC o generare un interrupt ePWM. Per le modalità up e down l'evento si verifica quando il valore scritto è uguale al $TBCTR$ una volta per ciclo, mentre per up-down due ad ogni periodo pwm come si vede in Fig 2.13.

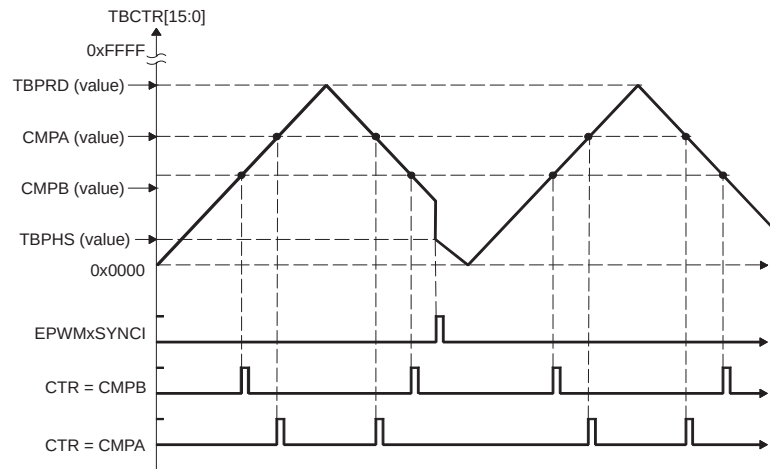


Figura 2.13. Eventi CMP ed impulso di SYNC

2.6.3 AQ

L'*Action Qualifier* specifica le azioni da compiere quando intervengono gli eventi dei sottomoduli time-base, counter-compare e trip-zone: tramite software si possono imporre le uscite EPWMxA e EPWMxB in modo che non siano modificate, siano impostate al valore *high*, *low* oppure che avvenga un *toggle*.

Lo scopo del programma è quello di ottenere le forme d'onda di Fig 2.14, e per questo i registri vengono impostati come segue:

`EPwm1Regs.AQCTLA.bit.CAU=AQ_SET` quando `TBCTR=CMPA` nella parte di conteggio Up l'uscita EPWM1A è alta;

`EPwm1Regs.AQCTLA.bit.CBD=AQ_CLEAR` quando `TBCTR=CMPB` nella parte di conteggio Down l'uscita EPWM1A è bassa;

`EPwm2Regs.AQCTLA.bit.CBU=AQ_SET` quando `TBCTR=CMPB` nella parte di conteggio Up l'uscita EPWM2A è alta;

`EPwm2Regs.AQCTLA.bit.CAD=AQ_CLEAR` quando `TBCTR=CMPA` nella parte di conteggio Down l'uscita EPWM2A è bassa;

ed infine il modulo ePWM4, un'onda quadra che servirà all'*Event Trigger* per avviare l'ADC, è configurato con `CMPA=Tpwm/4` e rimane fisso.

2.6.4 DB

Poiché i dispositivi di commutazione commutano in un tempo non nullo, è necessario inserire un ritardo tra lo spegnimento di un dispositivo e l'accensione dell'altro della stessa gamba per prevenire un cortocircuito.

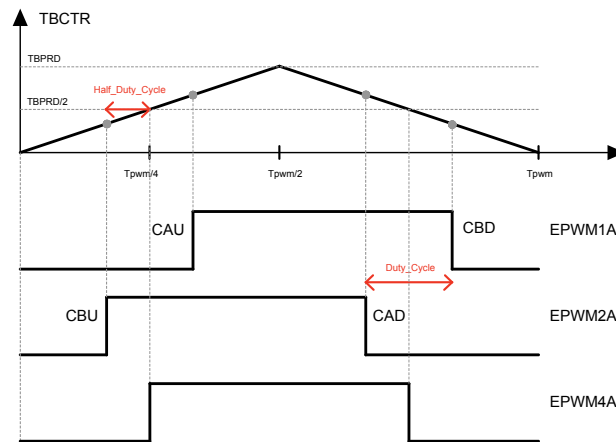


Figura 2.14. Action qualifier ed uscite PWM

Il *Dead-Band Generator* è in grado di generare due segnali, EPWMxA e EPWMxB, che sono mutuamente esclusivi e hanno un ritardo nei fronti l'uno rispetto all'altro.

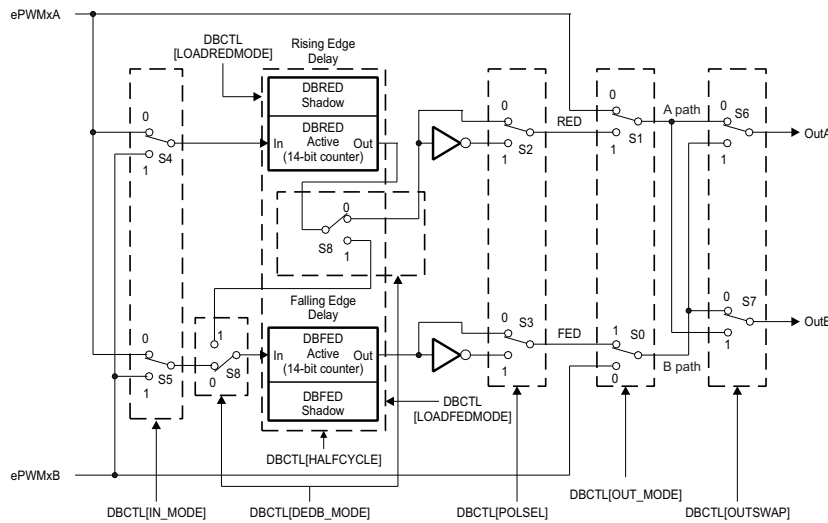


Figura 2.15. Configurazione sottomodulo Dead Band

I registri vengono così impostati:

DBCTL EPwmxRegs.DBCTL.bit.IN_MODE=0 il segnale EPWMxA proveniente dal AQ è la sorgente di ingresso per entrambi i ritardi sui fronti;
 EPwmxRegs.DBCTL.bit.POLSEL=2 imposta *Active High Complementary* (AHC), con gli switch in Fig 2.15 posti S3=1 e S2=0, ed il segnale EPWMxB risulta dunque invertito rispetto a EPWMxA;

`EPwmxRegs.DBCTL.bit.OUT_MODE=3` abilita il delay su entrambe le uscite, settando gli switch `S1=1` e `S0=1`, ovvero RED sul segnale `EPWMxA` e FED sul `EPWMxB`; il risultato delle configurazioni si può vedere in Fig 2.16;

`DBFED` e `DBRED` entrambi i registri impostano una costante definita nel file *Hardware_Interface.h* con `DEAD_TIME=53` e pari a $0.53\mu s$ dato il clock della pwm a 100MHz.

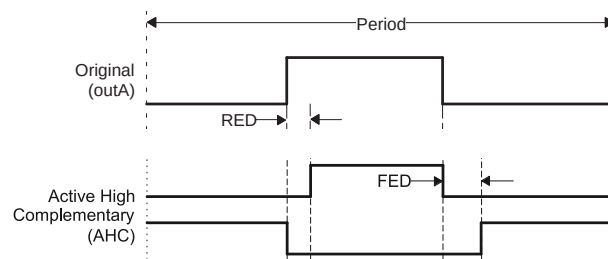


Figura 2.16. Forme d'onda DB

Grazie all'oscilloscopio è possibile acquisire i due fronti d'onda e verificare il tempo morto pari a $0.53\mu s$ in Fig 2.17.

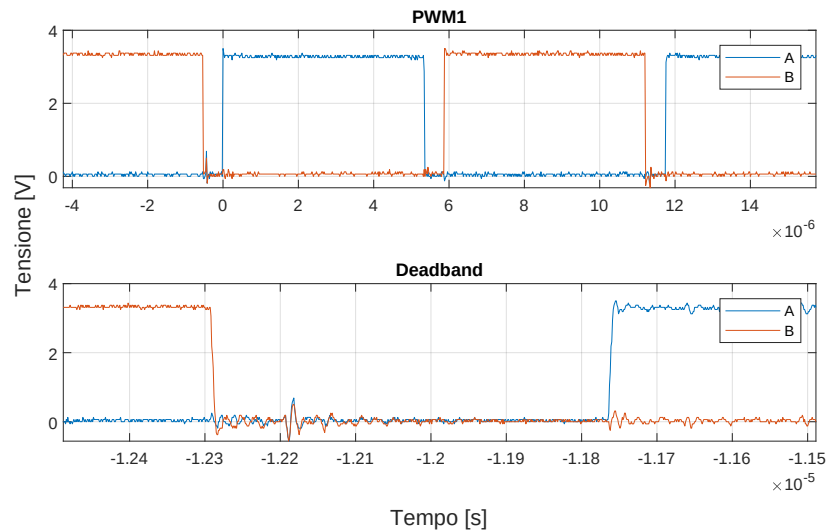


Figura 2.17. Deadband $0.53\mu s$ modulo ePWM1

2.6.5 ET

L'*Event Trigger* viene utilizzato nel programma per avviare l'ADC ogni quattro periodi di pwm quando $CTR = CMPB$ nel conteggio in decremento. I registri coinvolti:

`ETSEL EPwmxRegs.ETSEL.bit.SOCAEN=1` abilita l'impulso *ADC Start of Conversion A* (EPWMxSOCA);

`EPwmxRegs.ETSEL.bit.SOCASEL=7` l'evento avviene quando $TBCTR = CMPB$ e il contatore sta decrementando;

`ETPS.bit.SOCPSSEL=1` seleziona contatori `ETSOCPS [SOCACNT2/SOCBCNT2]`, e per determinare la frequenza degli eventi i registri `[SOCAPRD2/SOCBPRD2]` (impostabile da 0 a 15);

`ETSOCPS.bit.SOCAPRD2=4` imposta quanti eventi sono necessari prima di generare impulso SOCA;

queste impostazioni sono effettuate solo nel modulo **ePWM4A**, adibito alla funzione di trigger per SOC dell'ADC, e non dunque nella configurazione dei **ePWM1** e **ePWM2** che rimangono destinati alla gestione degli switch dell'inverter.

A dimostrazione di ciò, in Fig 2.18 vediamo il **TOGGLE GPIO22** inserito nell'interrupt associato all'ADC: ha una frequenza pari a un quarto di quella di PWM. Faccio notare come nel **TOGGLE** ad ogni ciclo la funzione va a cambiare lo stato del GPIO corrispondente; si deve dunque valutare il "semiperiodo", al quale si può notare corrispondono quattro periodi di pwm a 85kHz.

2.6.6 TZ

Il *Trip Zone* utilizza una logica veloce indipendente dal clock per gestire rapidamente i guasti forzando le uscite **EPWMxA** ed **EPWMxB** in uno stato sicuro, alto, basso o alta impedenza; si evita così qualsiasi latenza tramite ISR software, che potrebbe non proteggere l'hardware in condizioni di sovracorrente o cortocircuiti. Il trip zone supporta *one-shot trip* nel caso di guasti, ma anche il funzionamento *cycle-by-cycle trip* per limitazioni di corrente.

I segnali trip-zone sono generati **TZ1-3** tramite *Input X-Bar*, **TZ4** da un errore eQEP, **TZ5** da un errore del clock di sistema ed infine **TZ6** da un arresto di emulazione della CPU. Oltre a ciò, il sottomodulo *Digital Compare* può generare numerosi altri segnali trip-zone.

I registri usati sono:

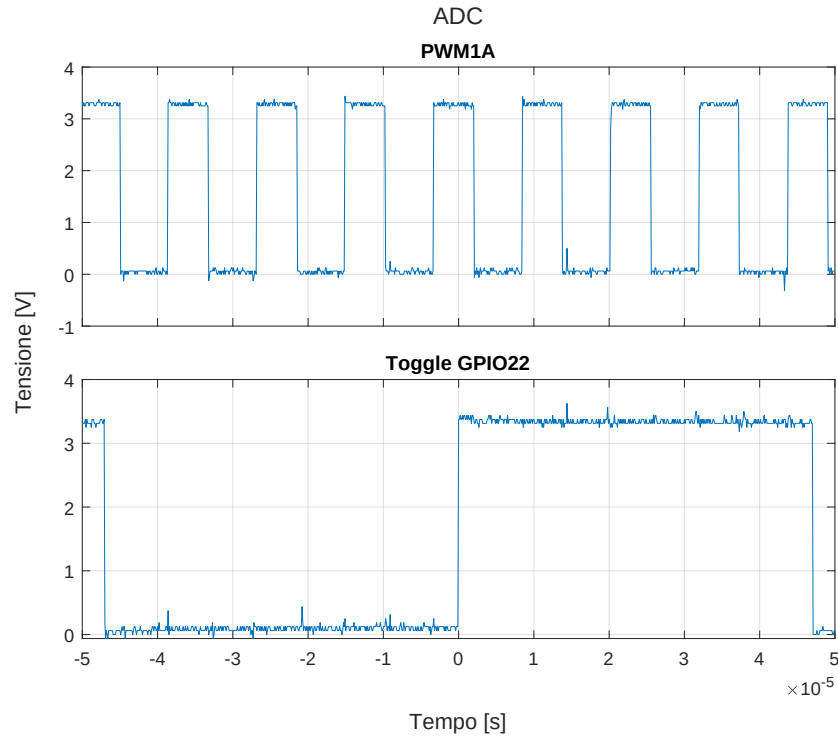


Figura 2.18. Interrupt ADC, TOGGLE GPIO22 con $f = 21,25kHz$

TZCTL *trip zone control*, imposta `EPwmxRegs.TZCTL.bit.TZA=2` e il registro `EPwmxRegs.TZCTL.bit.DCAEVT1=2` in modo da forzare EPWMxA allo stato *low* se i segnali TZ vanno allo stato basso (lo stesso per l'uscita B);

TZCTR *trip zone clear*, serve a pulire i flag `EPwmxRegs.TZCLR.bit.OST=1` del one-shot-trip, `EPwmxRegs.TZCLR.bit.INT=1` del trip-interrupt ed infine `EPwmxRegs.TZCLR.bit.DCx EVT1=1` per il segnale proveniente dal digital compare;

TZSEL *trip zone select*, abilita la sorgente one-shot-trip con il comando del registro `EPwmxRegs.TZSEL.bit.OSHTn=1` e per il segnale in arrivo dal digital compare `EPwmxRegs.TZSEL.bit.DCAEVT1=1`;

TZEINT *trip zone enable interrupt*, abilita l'interrupt relativo al one-shot-trip con `EPwmxRegs.TZEINT.bit.OST=1`.

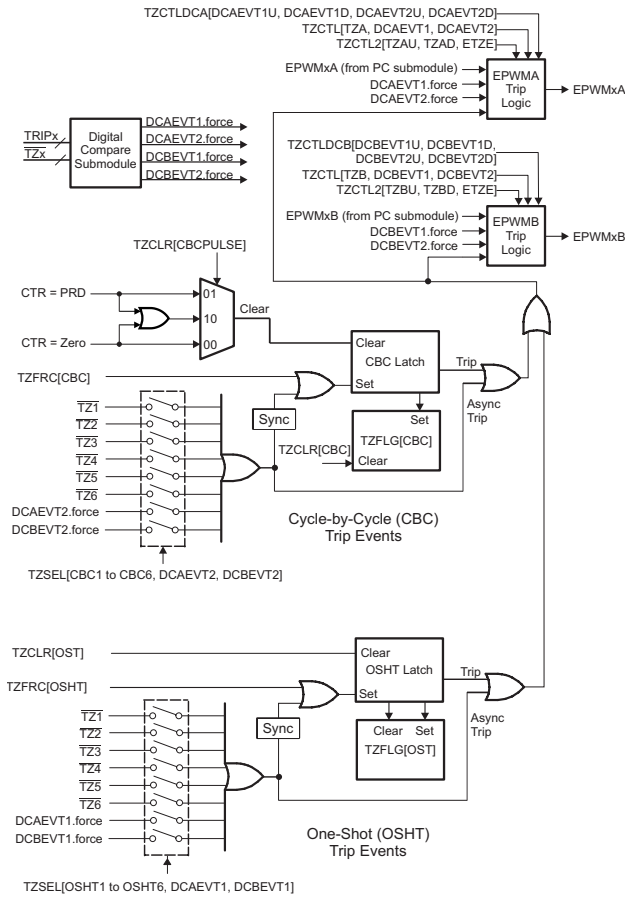


Figura 2.19. Trip Zone Submodule Mode Control Logic

I tre pin per TZ1-3 sono configurati attraverso il modulo *Crossbar* in Tab 2.2 e sono rispettivamente i GPIO25-27-14. Il quarto trip-zone, impostato sul pin GPIO15, viene configurato attraverso il sottomodulo *Digital Compare*.

In Fig 2.19 è possibile vedere la struttura del TZ, in particolare individuando due aree: nella parte superiore è presente il CBC cycle-by-cycle trip; in quella sottostante invece il OSHT one-shot trip, che viene effettivamente utilizzata all'interno del programma per salvaguardare i componenti del sistema.

2.6.7 DC

Il *Digital Compare* fornisce la possibilità di utilizzare una variegata quantità di segnali, individuabili in Fig 2.20, con lo scopo di ottenere dei comandi in direzione di tre sottomoduli. È necessario l'uso della *Crossbar* per la


```

    case CHOPPER:
        ...
        break;

505 case PHASE_SHIFT:
506     EALLOW;
507     //Time Base configuration
508     (*EPwmRegs_X).TBCTL.bit.FREE_SOFT=3;
509     (*EPwmRegs_X).TBCTL.bit.HSPCLKDIV = 0;
510     (*EPwmRegs_X).TBCTL.bit.CLKDIV = 0; //PWM Clock = 100MHz
511     (*EPwmRegs_X).TBCTL.bit.SYNCOSEL = 1;
512     (*EPwmRegs_X).TBCTL.bit.PHSEN=0;
513     (*EPwmRegs_X).TBCTL.bit.CTRMODE = 2; //Up-down mode
514     (*EPwmRegs_X).TBCTR = 0; //Clears the counter
515     (*EPwmRegs_X).TBPRD = Period; //PWM half period
516     (*EPwmRegs_X).TBPHS.all=0;
517
518     //Counter compare configuration
519     (*EPwmRegs_X).CMPCTL.bit.LOADAMODE=0;
520     (*EPwmRegs_X).CMPCTL.bit.LOADBMODE=0;
521     (*EPwmRegs_X).CMPA.bit.CMPA=0xFFFF;
522     (*EPwmRegs_X).CMPA.bit.CMPAHR=(1 << 8);
523     (*EPwmRegs_X).CMPB.bit.CMPB=0xFFFF;
524     (*EPwmRegs_X).CMPB.bit.CMPBHR=(1 << 8);
525
526     //Action qualifier configuration
527     if(Additional_Parameter==LEADING_PHASE){
528         (*EPwmRegs_X).AQCTLA.bit.CAD=AQ_CLEAR;
529         (*EPwmRegs_X).AQCTLA.bit.CBU=AQ_SET;}
530     else{
531         (*EPwmRegs_X).AQCTLA.bit.CBD=AQ_CLEAR;
532         (*EPwmRegs_X).AQCTLA.bit.CAU=AQ_SET;}
533
534     // PWM1B is the opposite of PWM1A
535     //Dead band generator configuration
536     (*EPwmRegs_X).DBCTL.bit.IN_MODE=0;
537     (*EPwmRegs_X).DBCTL.bit.POLSEL=2;
538     (*EPwmRegs_X).DBCTL.bit.OUT_MODE=3;
539     (*EPwmRegs_X).DBRED.bit.DBRED=DEAD_TIME; //rising edge
540     (*EPwmRegs_X).DBFED.bit.DBFED=DEAD_TIME; //falling edge
541
542     //Digital Compare configuration
543     //DCxH input select TRIP6
544     (*EPwmRegs_X).DCTRIPSEL.bit.DCAHCOMPSEL=5;
545     (*EPwmRegs_X).DCTRIPSEL.bit.DCBHCOMPSEL=5;
546     //DC output event DCxH = low, DCxL = don't care
547     (*EPwmRegs_X).TZDCSEL.bit.DCAEVT1=1;
548     (*EPwmRegs_X).TZDCSEL.bit.DCBEVT1=1;
549
550     //Trip Zone configuration
551     (*EPwmRegs_X).TZCTL.bit.TZA=2; //output low state
552     (*EPwmRegs_X).TZCTL.bit.TZB=2;
553     (*EPwmRegs_X).TZCTL.bit.DCAEVT1=2;
554     (*EPwmRegs_X).TZCTL.bit.DCBEVT1=2;
555     (*EPwmRegs_X).TZCLR.bit.OST=1; //Clears OST trip flag
556     (*EPwmRegs_X).TZCLR.bit.INT=1; //Clears INT flag
557     (*EPwmRegs_X).TZCLR.bit.DCAEVT1=1;
558     (*EPwmRegs_X).TZCLR.bit.DCBEVT1=1;
559     #ifndef DISABLE_TRIP_ZONE
560     (*EPwmRegs_X).TZSEL.bit.OSHT1=1;
561     (*EPwmRegs_X).TZSEL.bit.OSHT2=1;
562     (*EPwmRegs_X).TZSEL.bit.OSHT3=1;

```

```

563     (*EPwmRegs_X).TZSEL.bit.DCAEVT1=1;
564     (*EPwmRegs_X).TZSEL.bit.DCBEVT1=1;
565     (*EPwmRegs_X).TZEINT.bit.OST=1;
566     #else
567     (*EPwmRegs_X).TZSEL.bit.OSHT1=0;
568     (*EPwmRegs_X).TZSEL.bit.OSHT2=0;
569     (*EPwmRegs_X).TZSEL.bit.OSHT3=0;
570     (*EPwmRegs_X).TZSEL.bit.DCAEVT1=0;
571     (*EPwmRegs_X).TZSEL.bit.DCBEVT1=0;
572     (*EPwmRegs_X).TZEINT.bit.OST=0;
573     #endif
574
575     //Event trigger configuration
576     (*EPwmRegs_X).ETSEL.bit.SOCAEN=0;
577     (*EPwmRegs_X).ETSEL.bit.SOCBEN=0;
578     (*EPwmRegs_X).ETSEL.bit.INTEN=0;
579
580     //HRPWM
581     (*EPwmRegs_X).HRCNFG.all = 0x0;
582     (*EPwmRegs_X).HRCNFG.bit.EDGMODE = HR_BEP;
583     (*EPwmRegs_X).HRCNFG.bit.CTLMODE = HR_CMP;
584     (*EPwmRegs_X).HRCNFG.bit.HRLOAD = HR_CTR_ZERO_PRD;
585     (*EPwmRegs_X).HRCNFG.bit.EDGMODEB = HR_BEP;
586     (*EPwmRegs_X).HRCNFG.bit.CTLMODEB = HR_CMP;
587     (*EPwmRegs_X).HRCNFG.bit.HRLOADB = HR_CTR_ZERO_PRD;
588     #if(AUTOCONVERT)
589     (*EPwmRegs_X).HRCNFG.bit.AUTOCONV = 1; //Enable
590     #endif
591     (*EPwmRegs_X).HRPCTL.bit.HRPE = 0; //Turn off HRPRD
592     EDIS;
593     break;

```

```

    case DOM:
    ...
    break;

```

```

    case ADC_SOC:
    EALLOW;
    ...
    //Action qualifier configuration
    (*EPwmRegs_X).AQCTLA.bit.CAD=AQ_CLEAR;
    (*EPwmRegs_X).AQCTLA.bit.CAU=AQ_SET;
    ...
    //Event trigger configuration
    (*EPwmRegs_X).ETSEL.bit.SOCAEN=1; //triggers ADC
    (*EPwmRegs_X).ETSEL.bit.SOCASEL=7;
    (*EPwmRegs_X).ETSEL.bit.INTEN=0;
    #ifndef QUADRUPLER_SAMPLE_RATE
    (*EPwmRegs_X).ETPS.bit.SOCPSSEL=1;
    (*EPwmRegs_X).ETSOCPS.bit.SOCAPRD2=4; //forth event
    #else
    (*EPwmRegs_X).ETPS.bit.SOCAPRD=1;
    #endif
    (*EPwmRegs_X).ETCLR.bit.SOCA=1; //Clear SOCA flag
    EDIS;
    break;

```

```

    default:
    Result=FALSE;
}
return Result;
}

```

Le uscite dei moduli PWM si possono vedere in Fig 2.21, dove compaiono le forme d'onda dei moduli ePWM1A e ePWM2A quando lo sfasamento tra le due onde quadre è nullo; ePWM4A quando l'uscita è abilitata attraverso il PushButton. Faccio notare che quest'ultimo modulo in realtà è sempre in funzione, anche quando l'uscita corrispondente (GPI006) è disabilitata, poiché l'event trigger gestisce l'ADC_SOC.

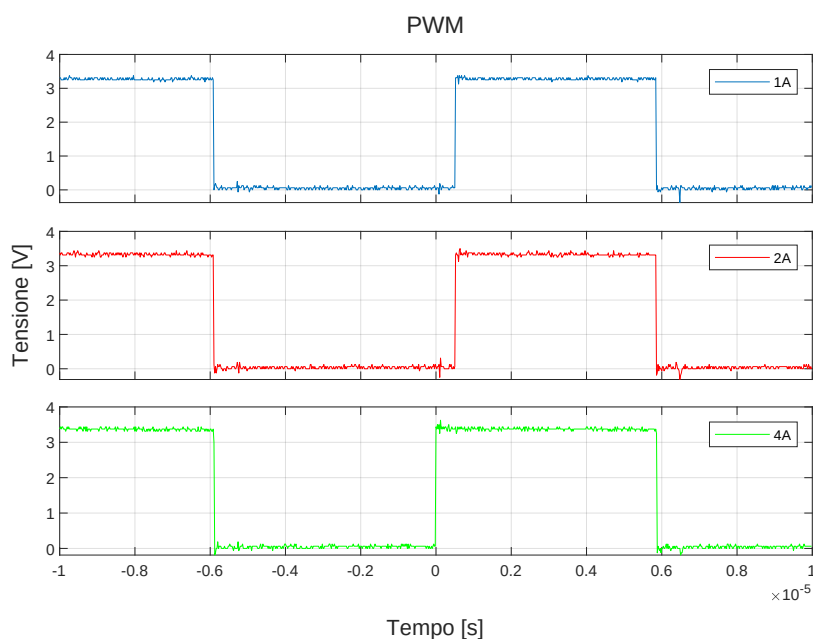


Figura 2.21. Uscite PWM

2.7 Analog to Digital Converter

Il modulo dell'ADC è sensibilmente diverso rispetto a quello del precedente microcontrollore: il dispositivo F28379D ne include infatti quattro indipendenti ad alte prestazioni a cui possono accedere entrambe le CPU. Ogni modulo ADC ha un singolo circuito di *sample-and-hold* (S/H), e utilizzandoli contemporaneamente è possibile eseguire un campionamento simultaneo. La periferica è implementata utilizzando un ADC ad approssimazione successiva (SAR) con una risoluzione programmabile: 16 bit con segnali differenziali, oppure 12 bit in modalità single-ended con un solo pin. Lo schema a blocchi si trova in Fig 2.22.

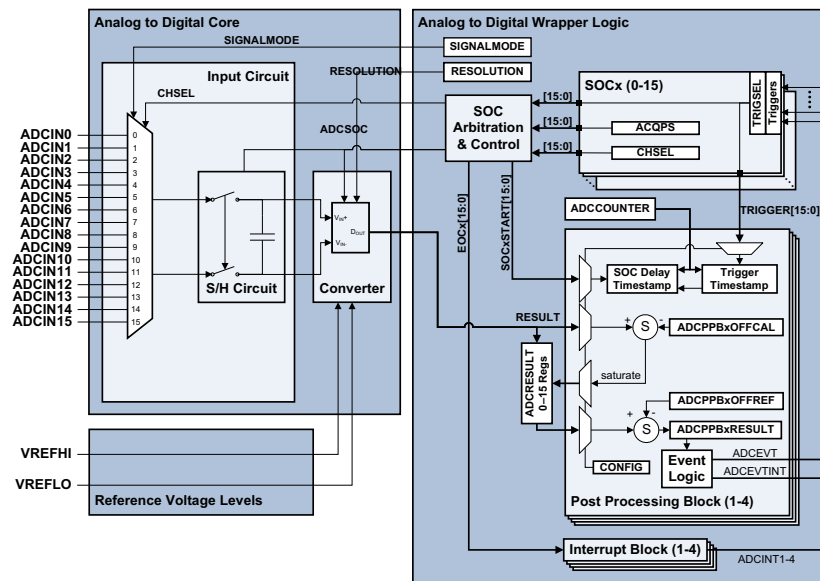


Figura 2.22. ADC schema a blocchi

L'ADC è composto da un **core** e un **wrapper**. Il *core* è composto dai *circuiti analogici* che includono il canale di selezione MUX, il sample-and-hold (S/H), i circuiti di approssimazione successiva, di riferimento tensione e altri di supporto analogici. Il *wrapper* è composto dai *circuiti digitali* che configurano e controllano l'ADC: la logica per conversioni programmabili, registri dei risultati, interfacce per circuiti analogici, per bus periferici, circuiti di post-elaborazione e interfacce per altri moduli on-chip.

Il trigger dell'ADC e il sequenziamento delle conversioni sono gestiti da una serie di registri *Start of Conversion* (SOCx); vedremo di seguito una spiegazione approfondita.

Dopo che ogni conversione di canale SOCx è stata completata, è possibile utilizzare un segnale *end of conversion* (EOC) per attivare un interrupt ADC: ogni modulo ha quattro interrupt configurabili (ADCINT1-4) che possono essere attivati da uno qualsiasi dei segnali EOC.

Per migliorare ulteriormente le capacità dell'ADC, ogni modulo ADC incorpora quattro *blocchi post-elaborazione* (PPB), collegati ai registri dei risultati, in grado di correggere offset, calcolo errore da un set-point, rilevamento di un limite e di passaggio per lo zero.

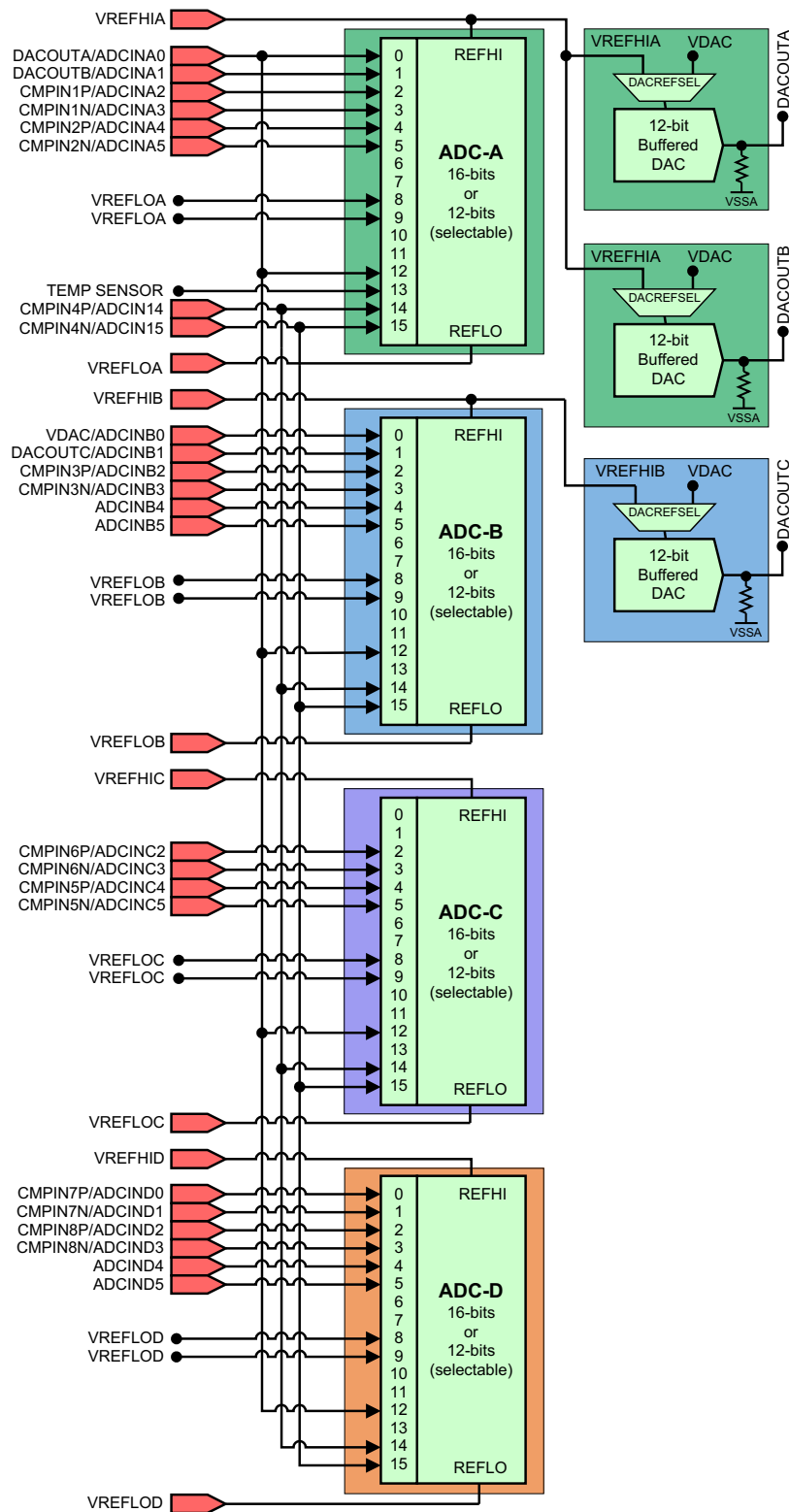


Figura 2.23. Analog subsystem (337-Ball ZWT)

Vediamo infine come è costituito l'*analog subsystem* in Fig 2.23. Gli schemi a blocchi del sottosistema mostrano le connessioni tra i diversi moduli analogici integrati ai pin del dispositivo (non tutti presenti nel LAUNCHXL-F28379D). Questi rientrano in due categorie: *ingressi/uscite* dei moduli analogici; *pin di riferimento*, da VREFHIA a VREFHID e da VREFLOA a VREFLOD, che possono essere utilizzati per fornire esternamente il riferimento all'ADC.

Nella scheda utilizzata sono presenti 16 pin ingresso per l'ADC: ADCINA0-5, ADCINB2-5, ADCINC2-5 ed infine ADCIN14-15. Nel connettore J21 in Fig 2.24 si trovano *ADC-D Differential Pair Inputs*, dei pin di collegamento per il modulo D, che includono ADCIND0-3. All'interno del programma verranno usati più pin di ingresso sfruttando i quattro moduli per avere più conversioni contemporaneamente.

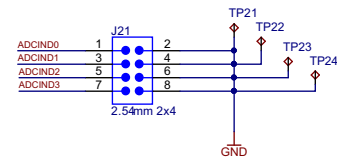


Figura 2.24

2.7.1 Configurazioni

Le configurazioni iniziali del modulo ADC sono eseguite all'interno del file `tx_Hardware_Interface.c` e sono riportate in Code 2.5.

Code 2.5. `tx_Hardware_Interface.c`

```

39  int Initialize_the_Microprocessor_Peripherals(){
    ...
235  /*      Initialize ADC      */
236  EALLOW;
237  AdcaRegs.ADCCTL2.bit.PRESCALE = 6;  //ADCCLK = 50 MHz
238  AdcbRegs.ADCCTL2.bit.PRESCALE = 6;
239  AdccRegs.ADCCTL2.bit.PRESCALE = 6;
240  AdcdRegs.ADCCTL2.bit.PRESCALE = 6;
241  AdcSetMode(ADC_ADCA, ADC_RES...12BIT, ADC_SIGNALMODE_SINGLE);
242  AdcSetMode(ADC_ADCB, ADC_RES...12BIT, ADC_SIGNALMODE_SINGLE);
243  AdcSetMode(ADC_ADCC, ADC_RES...12BIT, ADC_SIGNALMODE_SINGLE);
244  AdcSetMode(ADC_ADCD, ADC_RES...12BIT, ADC_SIGNALMODE_SINGLE);
245  AdcaRegs.ADCCTL1.bit.INTPULSEPOS=1; //Interrupt pulse at EOC
246  AdcbRegs.ADCCTL1.bit.INTPULSEPOS=1;
247  AdccRegs.ADCCTL1.bit.INTPULSEPOS=1;
248  AdcdRegs.ADCCTL1.bit.INTPULSEPOS=1;
249
250  AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;  //Accensione dell'ADC
251  AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1;
252  AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;
253  AdcdRegs.ADCCTL1.bit.ADCPWDNZ = 1;
254  DELAY_US(1000);  //Ritardo di 1ms
255  ...}

```

Il *clock* base in arrivo all'ADC è il SYSCLK del sistema, ed attraverso il registro `ADCCTL2.bit.PRESCALE` si determina l'ADCCLK, che definisce poi la finestra di acquisizione. Il clock massimo risulta di 50MHz (8.10.1.2.3 *ADC Operating Conditions, Microcontrollers datasheet* [5]).

Tabella 2.3. SOC canali

Segnale	Canali F28379D	Risultati AdcxResultRegs.	Pin Scheda controllo
Coil Current	ADCINA0	(a) ADCRESULT0	ADCA0
Envelope of Current	ADCINA1	(a) ADCRESULT1	ADCA1
Power Module Temperature	ADCIND0	(d) ADCRESULT0	ADCA3
Phase B Current	ADCINB2	(b) ADCRESULT0	ADCB0
Phase Displacement	ADCINB3	(b) ADCRESULT1	ADCB1
DC Bus Voltage	ADCINC2	(c) ADCRESULT0	ADCA2
Digital Supply Voltage	ADCINC3	(c) ADCRESULT1	ADCB2

La *risoluzione* si può scegliere tra 16 bit e 12 bit, ed è consigliabile configurare la modalità attraverso la function `AdcSetMode` fornita in C2000ware nel file `F2837xD_Adc.c`.

La *sequenza di accensione* prosegue quindi, dopo aver abilitato il clock al modulo con `CpuSysRegs.PCLKCR13.bit.ADC_x=1` e settato il `PRESCALE`, con l'accensione `ADCPWDNZ` e un ritardo finale.

2.7.2 SOC

Il trigger e il sequenziamento di conversione è ottenuto attraverso *Start of Conversion* (SOCn), un set di tre configurazioni che definisce la singola conversione di un singolo canale. In ogni SOC sono presenti:

- *sorgente di trigger*, avvia la conversione, è impostato su `ADCSOCA` del modulo ePWM4; il registro è `AdcxRegs.ADCSOCnCTL.bit.TRIGSEL`
- *canale* da convertire, espresso in `AdcxRegs.ADCSOCnCTL.bit.CHSEL`. In Tab 2.3 si vedono i canali e i relativi segnali. Da notare che il canale e il nome del pin sulla scheda di controllo non corrispondono a causa del differente architettura presente nel vecchio microcontrollore.
- *durata dell'acquisizione*, la finestra è definita $= (ACQPS + 1) \cdot T_{SYSCLK}$; la durata minima deve essere pari ad almeno un ciclo di `ADCCLK`, e non minore a 75ns (vedere datasheet [5]). Il registro corrispondente è `AdcxRegs.ADCSOCnCTL.bit.ACQPS`

In Fig 2.25 si nota la configurazione di uno dei sedici SOC presenti.

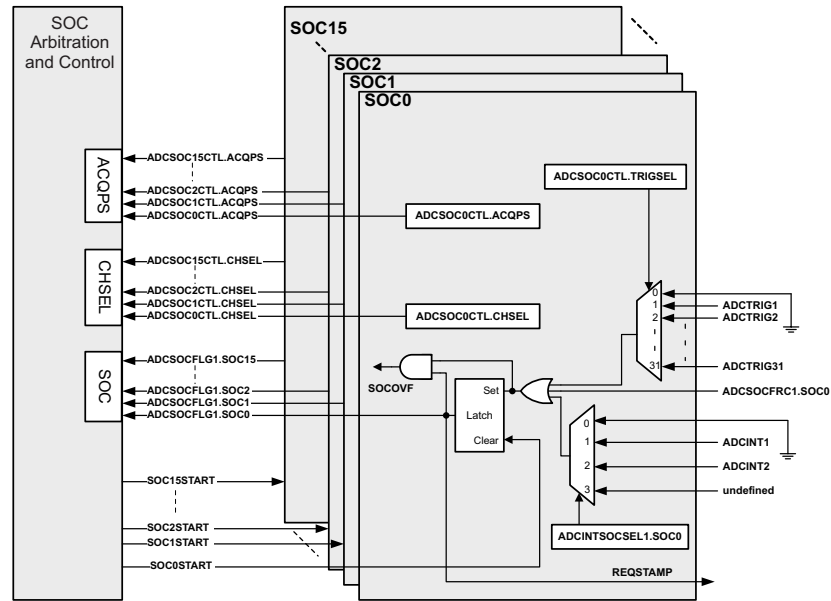


Figura 2.25. Schema a blocchi SOC

2.7.3 Sincronizzazioni tra moduli

Per garantire il funzionamento sincrono, tutti gli ADC sul dispositivo devono funzionare in *lockstep*: ciò è ottenuto impostando i registri in modo che fasi di *campionamento* e *conversione* siano allineate; il modo più semplice è scrivere nei SOC uguali TRIGSEL trigger e ACQPS finestra di acquisizione (durata S/H). La fase di *conversione* a 12 bit ha un'estensione minima di 10.1 e massima di 11 cicli ADCCLK (vedere datasheet [5]).

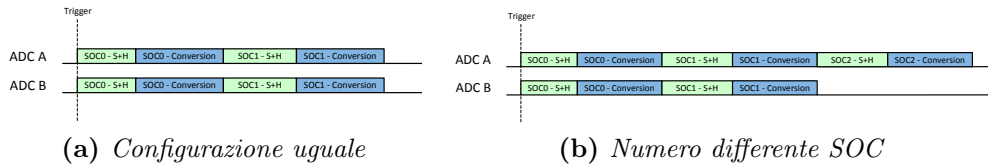


Figura 2.26. ADC funzionamento sincrono tra moduli

Il risultato ottenuto in Fig 2.26a prevede l'uso dei due moduli ADCA e ADCB, i cui SOC sono stati impostati in modo da avere stesso trigger e valori identici di ACQPS per SOC con ugual numero. Possono quindi essere utilizzate finestre di acquisizione diverse ma queste devono risultare le stesse per canali campionati simultaneamente.

Il funzionamento all'interno del programma prevede tuttavia l'uso di un numero differente di SOC: in Fig 2.26b si vede che i moduli rimangono sincroni fintanto che non si ha un overflow del trigger.

2.8 SPI e SCI

La SPI è una porta seriale sincrona ad alta velocità che trasporta un flusso di bit seriale di lunghezza programmabile da e verso il dispositivo a una velocità di trasferimento dei bit programmabile. Viene in genere utilizzata per le comunicazioni tra processori e periferiche esterne.

La SCI è una porta seriale asincrona (nota anche come UART) che supporta le comunicazioni tra il processore e altre periferiche asincrone che utilizzano il formato standard di non ritorno a zero (NRZ).

2.8.1 Serial Peripheral Interface

In Fig 2.27a si possono vedere gli elementi principali del modulo SPI, tra cui i segnali:

SPICLK il *clock*, è approfondito di seguito;

SPISIMO slave in, master out;

SPISOMI slave out, master in;

SPISTE *slave transmit enable*, se attivo impedisce allo slave di ricevere dati;

il clock che arriva alla periferica è pari a LSPCLK, ovvero 50MHz per le configurazioni iniziali.

Viene definito quindi un *master*, il microcontrollore, che fornisce il SPICLK, e uno *slave*; per entrambi i dati vengono spostati fuori dai *shift register* su un fronte del clock e altri dati registrati nel fronte opposto. In questo modo, i due controller inviano e ricevono dati simultaneamente.

In modalità master (`MASTER_SLAVE=1`) i dati vengono emessi dal pin SPISIMO e ricevuti dal pin SPISOMI. Il buffer di trasmissione SPITXBUF invia i dati al registro SPIDAT, che a sua volta avvia la trasmissione sul pin SPISIMO, partendo dal MSB (most significant bit). Contemporaneamente, i dati ricevuti vengono spostati attraverso il pin SPISOMI nel LSB (least significant bit) di SPIDAT. Quando il numero di bit selezionato è stato trasmesso, i bit ricevuti sono trasferiti al buffer SPIRXBUF, archiviandoli in modo giustificato a destra.

Il formato del dato scrivendo `SpiaRegs.SPICCR.bit.SPICHAR=7` è imposto 8 bit per word. Si possono scrivere fino a 16 bit, e i dati quando inseriti

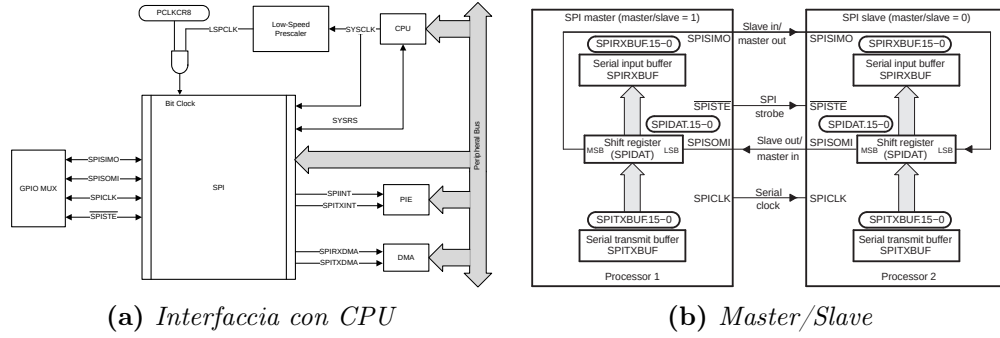


Figura 2.27. SPI schema a blocchi

in SPITXBUF e SPIDAT sono giustificati a sinistra, quando letti da SPIRXBUF giustificati a destra; quest'ultimo registro contiene dunque gli ultimi caratteri ricevuti, e in più i bit che rimangono dalla precedente trasmissione shiftati a sinistra.

Il registro SPIBRR determina il *bit transfer rate* per la rete sia in trasmissione che in ricezione:

$$\text{se } \text{SPIBRR}=0,1,2 \quad \text{SPI}_{\text{Baud Rate}} = \frac{\text{LSPCLK}}{4} \quad (2.4)$$

$$\text{se } \text{SPIBRR}=3-127 \quad \text{SPI}_{\text{Baud Rate}} = \frac{\text{LSPCLK}}{\text{SPIBRR} + 1} = \frac{50\text{MHz}}{45 + 1} = 1.087\text{Mbps} \quad (2.5)$$

Sono presenti 4 *SPICLK Scheme* e si ottengono dalla combinazione di due registri; all'interno del programma vengono impostati:

`SpiaRegs.SPICCR.bit.CLKPOLARITY=0;`

`SpiaRegs.SPICTL.bit.CLK_PHASE=1;`

che garantiscono *Rising edge with delay*; la trasmissione dei dati avviene mezzo ciclo prima del fronte di salita del segnale SPICLK, la ricezione sul fronte di salita.

SPICLK è *simmetrico* solo quando il risultato di (SPIBRR+1) è un valore pari; se dispari e SPIBRR>3, il clock diventa *asimmetrico* e si palesano due eventualità: CLKPOLARITY=0 l'impulso low di SPICLK è un ciclo LSPCLK più lungo dell'impulso high; CLKPOLARITY=1 l'impulso high di SPICLK è un ciclo LSPCLK più lungo di quello low (Fig 2.28).

Infine i buffer di trasmissione e ricezione sono integrati ciascuno con un FIFO di 16 word. Il registro a 1 word SPITXBUF del SPI standard funge da buffer di transizione tra il TXFIFO e lo shift register; quando l'ultimo bit di quest'ultimo è stato spostato, il SPITXBUF verrà caricato nuovamente dal

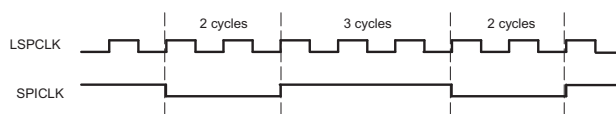


Figura 2.28. SPICLK asimmetrico, $(BRR+1)$ dispari e CLKPOLARITY=1

TXFIFO. Con il FIFO a 16 livelli è dunque possibile ridurre il carico generale sulla CPU.

2.8.1.1 Esempio trasmissione dati

Il diagramma temporale mostrato nella Fig 2.29 illustra un trasferimento di dati SPI tra due dispositivi utilizzando una lunghezza di cinque bit per carattere con SPICLK simmetrico ($BRR+1$ è pari). Il master scrive il valo-

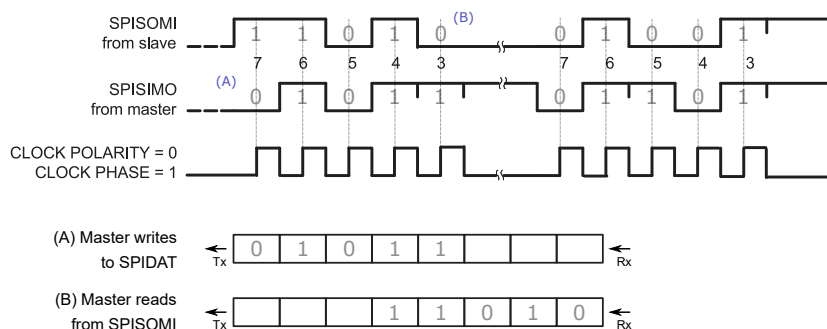


Figura 2.29. SPI cinque bit per Char

re $58h(0101\ 1000b)$ dal buffer SCITXBUF al registro SPIDAT; la trasmissione del carattere termina quando i 5 bit sono stati shiftati dal registro. A questo punto il *char* è stato inviato dallo slave completamente, il master legge $1Ah(0001\ 1010b)$ dal buffer SPIRXBUF.

2.8.2 Serial Communications Interface

La SCI è una porta seriale asincrona che funziona tramite due segnali Rx e Tx, comunemente nota come UART (Universal Asynchronous Receiver/-Transmitter). Il protocollo utilizzato per il connettore sarà illustrato nel paragrafo 3.3 *Connessione RS232* del prossimo capitolo. Sia il trasmettitore che il ricevitore possiedono un FIFO a 16 livelli per ridurre il carico, ed ognuno ha i proprio bit di abilitazione e di interrupt; possono essere azionati indipendentemente per la comunicazione *half-duplex*, o simultaneamente

per la *full-duplex*. Inoltre il modulo verifica l'integrità dei dati tramite *break detection*, *parità*, *overrun* e gli *errore di frame*.

2.8.2.1 Architettura SCI

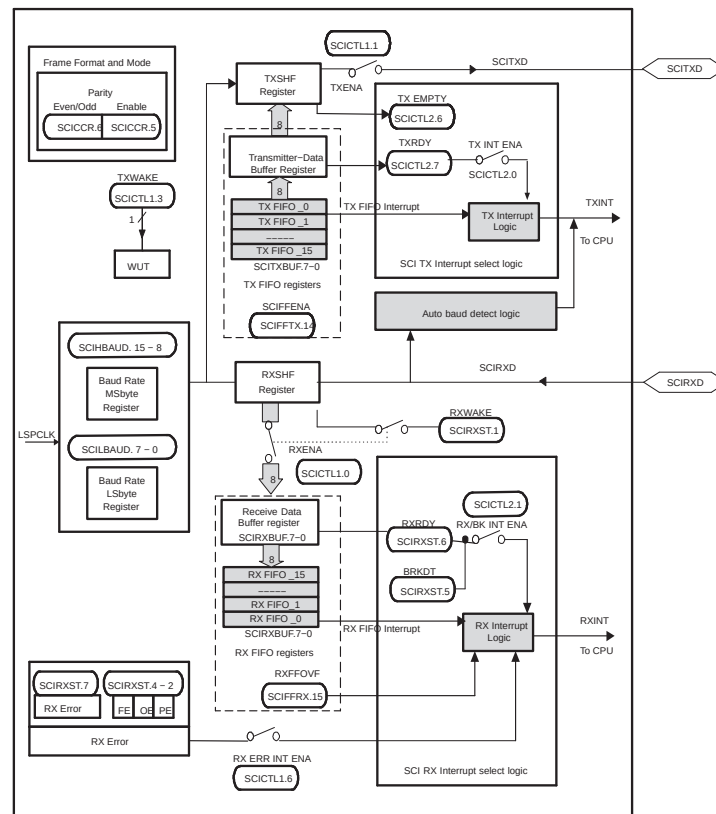


Figura 2.30. Serial Communications Interface (SCI)

In Fig 2.30 si può vedere l'architettura del modulo:

- un *trasmettitore* TX (nell metà superiore), i suoi registri principali
 - SCITXBUF, buffer che contiene i dati tx caricati dalla CPU da trasmettere;
 - TXSHF, shift register che riceve i dati dal buffer SCITXBUF ed invia i bit sul pin SCITXD uno alla volta;
- un *ricevitore* RX (nella metà inferiore), i suoi registri principali
 - RXSHF, shift register che riceve un bit alla volta dal pin SCIRXD;

- SCIRXBUF, buffer dei dati rx che devono essere letti dalla CPU;
- *baud generator* programmabile

$$\text{SCI}_{\text{Asynchronous Baud}} = \frac{\text{LSPCLK}}{(\text{BRR} + 1) \cdot 8} \quad \text{con BRR} = 1 - 65536 \quad (2.6)$$

i moduli SCIA e SCIB sono impostati rispettivamente a 38.58kHz e 102.459kHz;

- registri di controllo e stato.

2.8.2.2 Struttura del frame

I dati ricevuti ed inviati sono nel formato NRZ (non-return-to-zero), mostrato in Fig 2.31a, ed è composto da:

- un *bit start*;
- da uno fino a otto *data bit*, nel programma `SCICCR.bit.SCICHAR=7` impostiamo il massimo;
- un *bit di parità* pari o dispari; nel file `SCICCR.bit.PARITYENA=1` abilita la verifica, `SCICCR.bit.PARITY=0` sceglie la dispari;
- uno o due *stop bit*, nel programma `SCICCR.bit.STOPBITS=0` ne imposta uno;
- un *address bit* per distinguere gli indirizzi dai dati.

L'unità base dei dati è chiamata *character* ed è lunga da 1 a 8 bit; se a questo si aggiungono le informazioni di formattazione (bit start, stop, parità) otteniamo il *frame*.

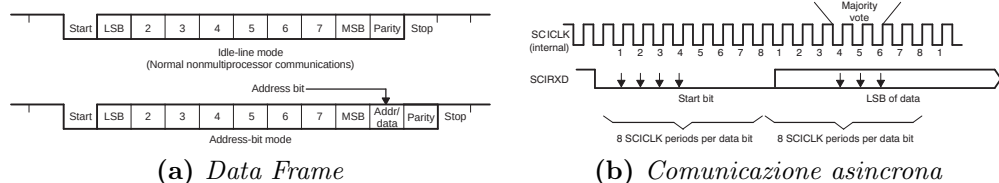


Figura 2.31. SCI Format

La comunicazione SCI asincrona utilizza otto periodi SCICLK (clock interno) per *data bit*, come illustrato in Fig 2.31b. Il ricevitore inizia a funzionare

alla ricezione di un *bit di start* valido, che è identificato da quattro periodi consecutivi di **SCICLK** con bit a zero. I bit successivi vengono determinati facendo tre campionamenti a metà del bit nel quarto, quinto e sesto periodo; il valore è determinato per maggioranza (due su tre).

Capitolo 3

Aggiornamento software

In questo capitolo verranno trattati dei miglioramenti al sistema di controllo per il WPT, come l'uso dei moduli pwm ad alta risoluzione e, più importante, l'impostazione di entrambi i core presenti nel TMS320F28379D. Verranno citati anche dei miglioramenti futuri.

3.1 High Resolution PWM

Solo i moduli dal ePWM1 al ePWM8 hanno la possibilità di abilitare High Resolution PWM (vedere datasheet [5]). Questo modulo aumenta la risoluzione temporale, espandendo i registri associati ai CMP dell'uscita A e B, al TB *phase* ed infine al TB *period*; è anche possibile gestire con maggior precisione i tempi morti, ma questa funzione non è stata utilizzata. All'interno di questo primo paragrafo vedremo anche la logica del *micro edge positioner* (MEP), illustrando l'implementazione e il software di diagnostica self-check.

La *risoluzione* della pwm convenzionale si può calcolare:

$$\text{PWM}_{\text{risoluzione}}(\%) = \frac{f_{\text{PWM}}}{f_{\text{EPWMCLK}}} \cdot 100\% = \frac{85\text{KHz}}{100\text{MHz}} \cdot 100 = 0.085\%$$
$$\text{PWM}_{\text{risoluzione}}(\text{bits}) = \log_2 \frac{T_{\text{PWM}}}{T_{\text{EPWMCLK}}} = \log_2 \frac{100\text{MHz}}{85\text{KHz}} = 10.2 \text{ Bits}$$

e in Fig 3.1 si vede la scala dei tempi.

Il modulo HRPWM è basato sul funzionamento del MEP: la logica è in grado di posizionare un edge in modo molto preciso suddividendo un *system clock* in più parti, arrivando a step minimi di 150ps. La Fig 3.2 riporta la relazione tra gli step derivati dal system clock e quelli del MEP; per definirli

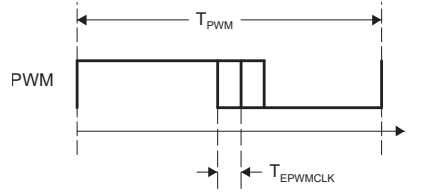


Figura 3.1. Risoluzione PWM convenzionale

è necessario calcolare:

$$N^{\circ}\text{step EPWMCLK} = \text{parte intera}(Duty_{PWM} \cdot Period_{PWM})$$

$$N^{\circ}\text{step MEP} = \text{parte frazionaria}(Duty_{PWM} \cdot Period_{PWM}) \cdot (MEP_{\text{scale factor}}) + 0.5$$

$$CMPA \text{ (16-bit)} = N^{\circ}\text{step EPWMCLK}$$

$$CMPAHR \text{ (16-bit)} = (N^{\circ}\text{step MEP}) \ll 8$$

La risoluzione massima ottenibile alla frequenza di PWM di 85kHz, considerando step minimi di 150ps, risulta pari a 16,2Bits.

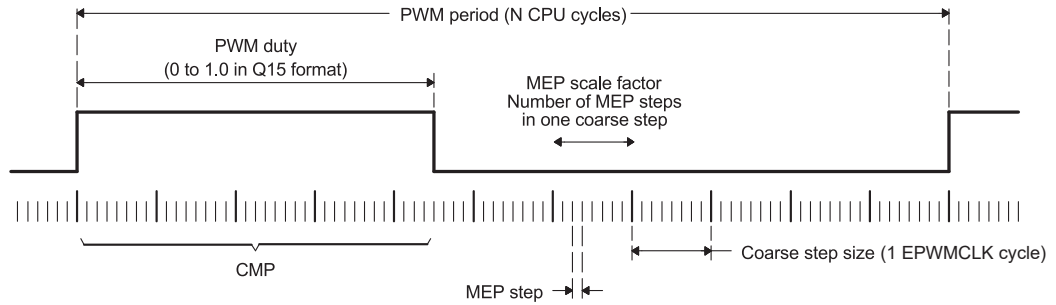


Figura 3.2. Logica funzionamento MEP

Dopo la configurazione iniziale dei registri, vedremo come impostare correttamente il duty cycle e i due registri da 16 bit relativi a CMPx e CMPxHR.

3.1.1 Configurazione

La configurazione avviene scrivendo nei registri HRCNFG, andando ad impostare i vari *Mode*:

- EDGMODE il MEP può essere programmato per fornire controllo di posizione preciso nell'edge salita (RE), in quello di discesa (FE) od in entrambi (BE);

- **CTLMODE** il MEP può essere controllato con i registri **CMP** per il controllo del duty cycle o con **TBPHS** per il controllo di fase;
- **HRLOAD** seleziona l'evento di tempo che carica il valore dal registro shadow a quello attivo;
- **AUTOCONV** è utilizzato insieme al software SFO *scale factor optimization*; se il registro è abilitato, il compare sarà $\text{CMPAHR} = \text{parte frazionaria}(\text{PWMduty} \cdot \text{PWMperiod}) \ll 8$ e il MEP *scale factor* è calcolato in background dal SFO e aggiorna automaticamente il registro **HRMSTEP** con il numero di MEP step per step grossolani; se è disabilitato l'utente dovrà impostare $\text{CMPAHR} = (\text{parte frazionaria}(\text{PWMduty} \cdot \text{PWMperiod}) \cdot \text{MEP}_{\text{scale factor}} + 0.5) \ll 8$

La parte di codice è riportata in Code 3.1.

Code 3.1. Hardware_Interface.c

```

612 //HRPWM
613 (*EPwmRegs_X).HRCNFG.all = 0x0;
614 (*EPwmRegs_X).HRCNFG.bit.EDGMODE = HR_BEP; //both edges
615 (*EPwmRegs_X).HRCNFG.bit.CTLMODE = HR_CMP;
616 (*EPwmRegs_X).HRCNFG.bit.HRLOAD = HR_CTR_ZERO_PRD;
617 (*EPwmRegs_X).HRCNFG.bit.EDGMODEB = HR_BEP; //both edges
618 (*EPwmRegs_X).HRCNFG.bit.CTLMODEB = HR_CMP;
619 (*EPwmRegs_X).HRCNFG.bit.HRLOADB = HR_CTR_ZERO_PRD;
620 #if(AUTOCONVERT)
621 (*EPwmRegs_X).HRCNFG.bit.AUTOCONV = 1; //enable
622 #endif
623 (*EPwmRegs_X).HRPCTL.bit.HRPE = 0; //turn off high-resolution period control

```

Per la definizione del duty cycle verrà utilizzata, come già detto, l'autoconversione e l'uso del software SFO. La dimensione del passo MEP varia in base ai parametri di processo, alla temperatura di esercizio e alla tensione; la funzione **SFO()** fornita dalla TI permette di determinare dinamicamente il numero di MEP step per ciascun periodo di **EPWMCLK** mentre il modulo **HRPWM** è in funzione.

Il valore ottimale del MEP scale factor viene determinato dalla funzione di self-check e diagnostica interna al modulo **HRPWM** stesso. Attraverso le librerie fornite, è possibile definire il valore corretto per le condizioni operative.

Nel caso di **EPWMCLK** = 100MHz e assumendo MEP step pari a 150ps, il tipico *scale factor* risulta di 66 MEP step per unità **TBCLK** (10ns).

La funzione SFO deve essere inizializzata durante il primo avvio del microcontrollore, per poi essere richiamata periodicamente al fine di aggiornare i valori del **HRMSTEP**. Se durante il processo c'è un errore, viene

riportato `SFO_ERROR` e il programma entra nella modalità allarme. Di seguito viene riportato il Code 3.2 che evidenzia la posizione di `SFO()` nel file *Control_Main.c*

Code 3.2. Control_Main.c

```

93 while(1){
94   if(Time_Interval_of_1ms_Elapsed==TRUE){
95     ...
128     switch(State_of_the_Management_Algorithm)
129     {
130       case STATE_MACHINE_INITIALIZATION:
131         Result&=Initialize_the_Software();
132         Result&=Initialize_the_Hardware();
133         while(status == SFO_INCOMPLETE){
134           status = SFO();
135           if (status == SFO_ERROR){
136             State_of_the_Man...=INITIALIZATION_ALARM;}
137         }...
138         break;
139         ...
140       case OPERATIONAL:
141         ...
323         status = SFO(); //continuously updates
324         if(status == SFO_ERROR){
325           State_of_the_Man...=INITIALIZATION_ALARM;}
326         ...
327         break;
328         ...
329       }//switch
330 }//if(Time_Interval_of_1ms_Elapsed==TRUE)
415 }//while(1)

```

Come si può notare la funzione è richiamata nello `switch`, in particolare all'interno di `STATE_MACHINE_INITIALIZATION` dopo l'inizializzazione dell'hardware; l'aggiornamento continuo avviene solo durante il funzionamento `OPERATIONAL`.

3.1.2 CMPHR

Il registro CMPA (e similmente il CMPB) è composto da 32 bit totali, 16 dei quali dedicati all'alta risoluzione. Come si vede in Fig 3.3 la composizione permette di allocare le due informazioni in un'unica soluzione o in modo separato. Nel codice a seguire si è preferito operare in maniera divisa, e la conversione avviene attraverso la logica `AUTOCONV` vista in precedenza.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPA																CMPAHR															
R/W-0h																R/W-0h															

Figura 3.3. Registro CMPA

In Code 3.3 è riportata la parte di codice che si trova all'interno della routine di servizio dell'interrupt, che calcola ed imposta i registri. L'elemento chiave è ovviamente la variabile `Half_of_Duty_Cycle`, che esprime metà del valore del duty cycle in `float`: data la tipologia del dato, è possibile scindere una parte *intera* ed una *frazionaria*, che andranno a comporre rispettivamente il `CMPx` e il `CMPxHR`.

Per una praticità di scrittura e di visualizzazione, le variabili sono state così rinominate: quarto di periodo `QPWMPRD = Quarter_of_PWM_Period`, metà del duty cycle `HDC = Half_of_Duty_Cycle`.

Code 3.3. `tx_Interrupt_Service_Routines.c`

```

39  interrupt void ADC_isr(){
...
205  if((State_of_the_Ma...==OPERATIONAL)&&(Blocking_Al...==0)){
206      if(PWM_Enabled==FALSE){
207          PWM_Enabled=Enable_PWM();}
208      EALLOW;
209      ...
251      Half_of_Duty_Cycle=Duty_Cycle/2;
252
253      PERIOD_PHASE_A=(int)Half_of_PWM_Period;
254      PERIOD_PHASE_B=(int)Half_of_PWM_Period;
255      PERIOD_AUX_PWM=(int)Half_of_PWM_Period;
256
257      //HRPWM in Q16
258      DutyFine_UP=((QPWMPRD+HDC)-(int)(QPWMPRD+HDC))*65536;
259      DutyFine_DOWN=((QPWMPRD-HDC)-(int)(QPWMPRD-HDC))*65536;
260      ...
261      //ePWM1
262      DUTY...UP_PHASE_A=(int)(QPWMPRD+HDC); //Pwm1_CMPA
263      DUTY_CYCLE_UP_PHASE_A_HR=DutyFine_UP; //Pwm1_CMPAHR
264      if((int)DutyFine_UP==0){
265          DUTY_CYCLE_DOWN_PHASE_A=(int)(QPWMPRD-HDC); //Pwm1_CMPB
266      }
267      else{
268          DUTY_CYCLE_DOWN_PHASE_A=(int)(QPWMPRD-HDC+1); //Pwm1_CMPB
269      }
270      DUTY_CYCLE_DOWN_PHASE_A_HR=DutyFine_UP; //Pwm1_CMPBHR
271      //ePWM2
272      if((int)DutyFine_UP==0){
273          DUTY_CYCLE_UP_PHASE_B=(int)(QPWMPRD+HDC); //Pwm2_CMPA
274      }
275      else{
276          DUTY_CYCLE_UP_PHASE_B=(int)(QPWMPRD+HDC+1); //Pwm2_CMPA
277      }
278      DUTY_CYCLE_UP_PHASE_B_HR=DutyFine_DOWN; //Pwm2_CMPAHR
279      DUTY_CYCLE_DOWN_PHASE_B=(int)(QPWMPRD-HDC); //Pwm2_CMPB
280      DUTY_CYCLE_DOWN_PHASE_B_HR=DutyFine_DOWN; //Pwm2_CMPBHR
281      //ePWM4
282      DUTY_CYCLE_AUX_PWM_A=(int)QPWMPRD; //Pwm4_CMPA
283      DUTY_CYCLE_AUX_PWM_B=(int)HDC; //Pwm4_CMPB
284
285      EDIS;
286      }
287      ...
}

```

Nel codice riportato ci sono diverse cose da notare. Per le prove sul banco è stato definito `MANUAL_DUTY_CYCLE`, per poter definire il duty cycle tramite `External_Reference_1` impostato grazie ai pulsanti esterni (numero intero). Per una scelta di visualizzazione è stato impostato `Duty_Cycle = (float)External_Reference_1/4`; ad ogni aumento di un unità, il duty sarà incrementato di 0.25, mentre `Half_of_Duty_Cycle` di 0.125

Sono stati definiti `DutyFine` allo scopo di preservare l'informazione frazionaria, ed utilizzarla poi nei registri ad alta risoluzione. Seguendo le indicazioni presenti nelle librerie della TI, è possibile individuare come nel file `hrpwm_duty_sfo_v8.c` i dati di queste variabili siano gestiti in Q16 (16 bit per rappresentare la parte frazionaria); per passare a questo tipo di rappresentazione, partendo da base decimale, si deve shiftare di 16 elementi, oppure procedere come nell'esempio seguente

$$0.125_{(10)} \cdot 2^{16} = 8192_{(10)} = 0010000000000000_{(2)} \quad Q16 \text{ format}$$

I successivi passaggi sono maggiormente comprensibili seguendo l'approccio grafico in Fig 3.4: per il fronte di salita della EPWM1A è necessario calcolare la parte intera di $(T_{pwm}/4 + Duty/2)$ per ottenere `CMPA`; il `DutyFine_UP` è invece processato come la rimanente parte frazionaria, e corrisponde al `CMPAHR`. Al contrario per il fronte di salita di EPWM2A si procede al calcolo di `CMPB` con la parte intera di $(T_{pwm}/4 - Duty/2)$, mentre `CMPBHR` è pari al `DutyFine_DOWN`.

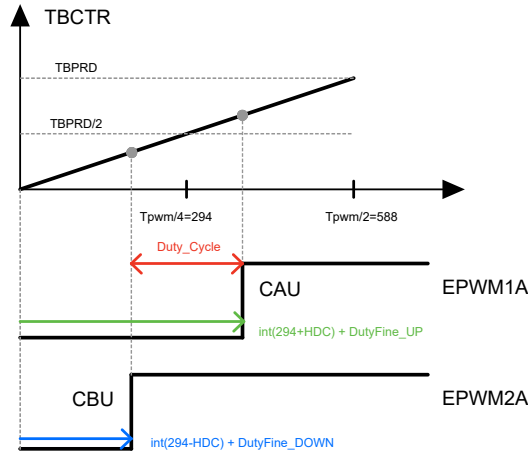


Figura 3.4. Principio funzionamento HRPWM

Vediamo dall'oscilloscopio un esempio per capire meglio quanto sopra illustrato. Impostando `External_Reference_1=1` otteniamo di conseguenza `Half_of_Duty_Cycle=0.125`; per i passaggi sopra riportati, i registri PWM

saranno:

$$\text{CMPA} = (\text{int})\left(\frac{T_{pwm}}{4} + \frac{Duty}{2}\right) = (\text{int})(294 + 0.125) = 294$$

$$\text{CMPAHR} = \text{DutyFine_UP} = 0.125 \cdot 65536 = 8192$$

$$\text{CMPB} = (\text{int})\left(\frac{T_{pwm}}{4} - \frac{Duty}{2}\right) = (\text{int})(294 - 0.125) = 293$$

$$\text{CMPBHR} = \text{DutyFine_DOWN} = 0.875 \cdot 65536 = 57344$$

In Fig 3.5 è possibile vedere lo step minimo di sfasamento pari a $10ns$ con il precedente software; con l'uso dell'alta risoluzione lo step diventa pari ad un quarto rispetto al caso precedente (step minimo in assoluto $150ps$, specifica datasheet).

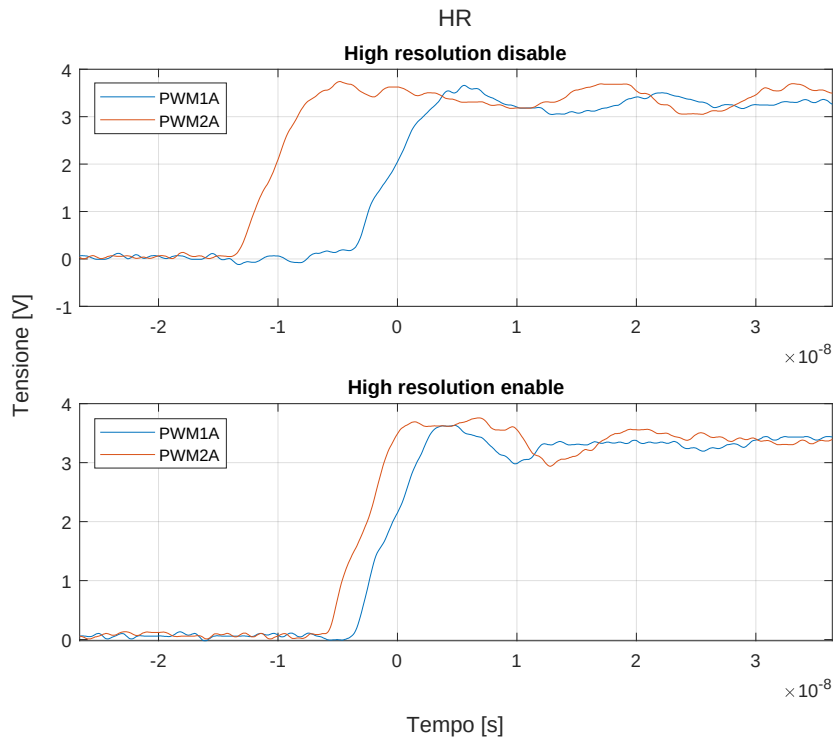


Figura 3.5. Step minimo, vantaggio prestazionale HRPWM

3.1.3 TBPRDHR e TBPHSHR

Oltre alla definizione di un registro ad alta risoluzione per il duty cycle (CMP), è possibile impostare anche HR per il periodo di pwm TBPRDHR e per lo sfasamento tra i due moduli attraverso TBPHSHR.

Nei paragrafi seguenti saranno illustrate due versioni differenti del software, una per ciascuna delle due funzionalità; entrambe presentano tuttavia delle problematiche per l'applicazione presa in considerazione.

Procediamo quindi con la configurazione del periodo pwm ad alta risoluzione, riportata in Code 3.4 con i principali registri. Bisogna procedere in ordine con i comandi:

1. abilitare clock ePWMx;
2. abilitare clock HRPWM, nella `function() Initialize_the_Peripherals_Clock;`
3. disabilitare TBCLKSYNC;
4. configurare i registri ePWMx (Aq, TBPRD, CC...);
5. configurare HRCNFG come visto in Code 3.1;
6. impostare HRPCTL[TBPHSHRLOADE]=1 e TBCTL[PHSEN]=1;
7. abilitare controllo del periodo HR HRPCTL[HRPE]=1;
8. abilitare TBCLKSYNC;
9. impostare TBCTL[SWFSYNC]=1;
10. impostare MEP scale factor attraverso function `SFO()`.

L'ultima operazione è quindi quella di scrivere direttamente nel registro ad alta risoluzione TBPRDHR.

Code 3.4. Hardware_Interface.c

```

419 int Initialize_PWMX(EPwmRegs_X, PWM_Method, Period, Add...Parameter){
420     switch(PWM_Method){
421         ...
505         case PHASE_SHIFT:
506             EALLOW;
507             CpuSysRegs.PCLKCRO.bit.TBCLKSYNC = 0;
423             ...
623             (*EPwmRegs_X).TBCTL.bit.PHSEN=1;
624             (*EPwmRegs_X).TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
625             (*EPwmRegs_X).TBCTL.bit.SYNCOSEL = TB_SYNC_DISABLE;
626             (*EPwmRegs_X).TBCTL.bit.HSPCLKDIV = TB_DIV1;
627             (*EPwmRegs_X).TBCTL.bit.CLKDIV = TB_DIV1;
628             (*EPwmRegs_X).TBCTL.bit.FREE_SOFT = 11;

```

```

...
693 (*EPwmRegs_X).HRCNFG.bit.AUTOCONV = 1;
694 (*EPwmRegs_X).HRPCTL.bit.TBPHSHRLOADE = 1;
695 (*EPwmRegs_X).HRPCTL.bit.HRPE = 1;
696 CpuSysRegs.PCLKCRO.bit.TBCLKSYNC = 1;
697 (*EPwmRegs_X).TBCTL.bit.SWFSYNC = 1;
698 EDIS
699 break;
...
870 return Result;
871 }

```

La definizione e il calcolo dei registri TBPRDHR avviene nella routine di servizio dell'interrupt dell'ADC, come si può vedere in Code 3.5: per ogni modulo pwm ne vengono utilizzati due, uno TBPRD tradizionale e il secondo ad alta risoluzione.

Code 3.5. tx_Interrupt_Service_Routines.c

```

39 interrupt void ADC_isr(){
...
239 PeriodFine=(Half_of_PWM_Period-(int)Half_of_PWM_Period)*65536;

240 PERIOD_PHASE_A=(int)Half_of_PWM_Period;
241 PERIOD_PHASE_A_HR= PeriodFine;
242 PERIOD_PHASE_B=(int)Half_of_PWM_Period;
243 PERIOD_PHASE_B_HR= PeriodFine;
244 PERIOD_AUX_PWM=(int)Half_of_PWM_Period;
245 PERIOD_AUX_PWM_HR=PeriodFine;
246 ...
}

```

La variabile `PeriodFine` viene calcolata nel formato Q16 e i procedimenti ripresi dal software `hrpwm_duty_sfo_v8.c` presente nelle librerie C2000.

Una variante prevede l'uso del TBPHS, permettendo uno sfasamento tra il modulo ePWM1 e ePWM2 pari al duty cycle. Operando in questo modo, è possibile definire i registri ad alta risoluzione come segue in Code 3.6.

Code 3.6. tx_Interrupt_Service_Routines.c

```

39 interrupt void ADC_isr(){
...
239 EPwm2Regs.TBPRDHR=PeriodFine;
240 EPwm2Regs.TBPRD=Period;
241 EPwm2Regs.TBPHS.bit.TBPHS=Duty_Cycle;
242 EPwm2Regs.TRREM.bit.TRREM=(Uint16)0x100+Duty_CycleFine; //Q8

243 EPwm1Regs.TBPRDHR=PeriodFine;
244 EPwm1Regs.TBPRD=Period;
245 EPwm1Regs.TRREM.bit.TRREM=0x100;
246 ...
}

```

L'impostazione dei registri è maggiormente comprensibile ispezionando il file fornito da TI `hrpwm_deadband_sfo_v8.c` nelle librerie C2000. Il registro TRREM (translator remainder) è utilizzato al posto di TBPHSHR (vedere nota 15.14.1.1 *Controlling the HRPWM Capabilities* del manuale [1]).

3.1.3.1 Limiti

Sono presenti ulteriori limiti per quanto riguarda l'uso del MEP: nella modalità ad alta risoluzione questo non è attivo per il 100% del periodo pwm. Quando TBPRDHR è abilitato, il MEP non è operativo nei primi tre cicli di EPWMCLK, nei tre cicli che precedono e seguono $CTR = PRD$, ed infine durante il conteggio down nei tre cicli prima di $CTR = 0$. Una migliore comprensione di ciò si può avere guardando la Fig 3.6.

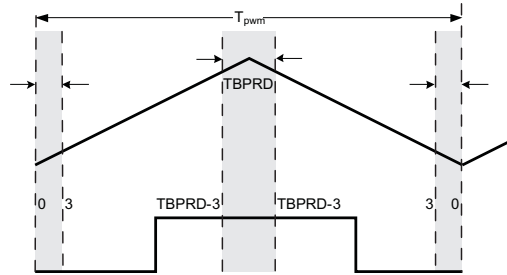


Figura 3.6. Limiti conteggio up-down

Le impostazioni relative alle pwm vedono i CMP posti in corrispondenza di $TBPRD/2$ quando $duty=0$. Da questo punto di vista il sistema non soffre dei limiti segnalati nella maggior parte del range; tuttavia il duty non potrà essere al massimo, lo sfasamento dovrà infatti estendersi nella regione consentita dal MEP.

3.1.3.2 Problemi riscontrati

A causa dei vincoli sopra descritti e al funzionamento intrinseco del MEP, durante lo sviluppo del codice sono state riscontrate molteplici difficoltà. I risultati ottenuti sono illustrati di seguito, presentando due problematiche differenti riscontrate per mezzo dell'oscilloscopio.

Una delle note presenti nella documentazione relativa (*15.14.1.5.4.1 High Resolution Period Configuration* del manuale [1]) specifica come sia necessario impostare $TBCTL[SYNCOSEL]=0$, altrimenti se si applica un *sync pulse* software apparirà un *jitter* pari a 2 cicli nel caso modalità up-down. Il problema nasce dal fatto che i primi due moduli pwm devono essere correlati per il tipo di applicazione. Se si ignora dunque l'avvertimento del produttore si ottiene il risultato in Fig 3.7 in cui appare evidente il fenomeno grazie alla persistenza sullo schermo dell'oscilloscopio.

Per risolvere questo errore, si procede con TBPHS ad alta risoluzione. L'uso comunque di questa funzione è quello per le applicazioni trifase, il cui sfasamento è di $2\pi/3$ per modulo. In questo funzionamento tuttavia, lo

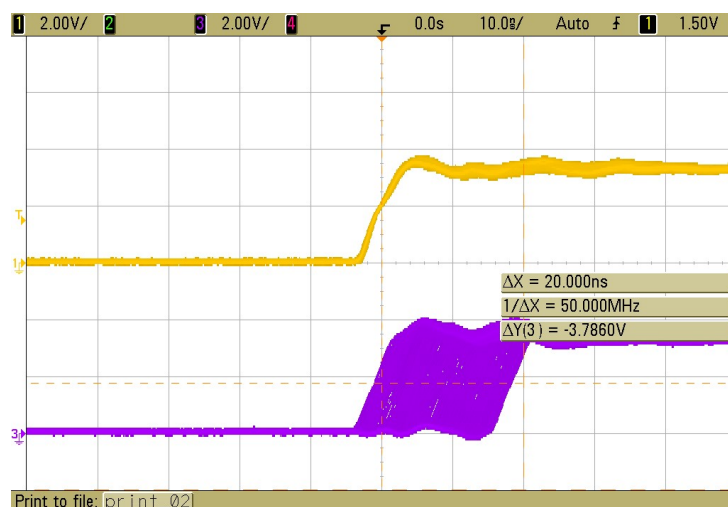


Figura 3.7. PWM1A in alto, PWM2A in basso

sfasamento diventa proporzionale al duty cycle (sfasamento tra le due onde quadre). L'alta risoluzione è quindi garantita, ma comprare uno sfasamento iniziale non voluto pari a due periodi e_{PWMCLK} come si può vedere in Fig 3.8.

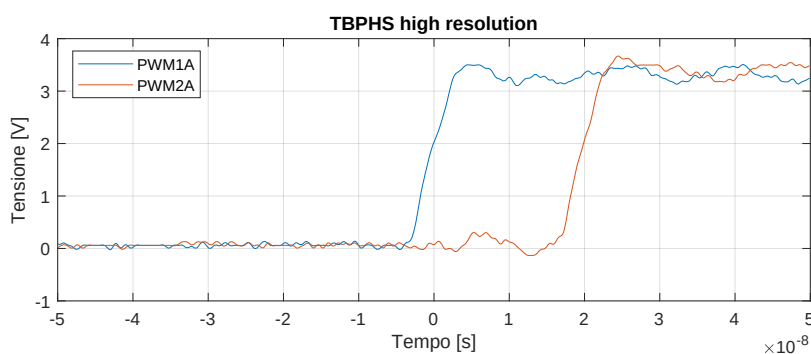


Figura 3.8. Problema sfasamento iniziale

Non rispettando gli obiettivi desiderati, si è scelto di non implementare l'uso di TBPRDHR e TBPHSHR.

Tabella 3.1. Registri IPC

Registro locale	CPU locale	CPU remota	Registro remoto
IPCSENDCOM	R/W	R	IPCRECVCOM
IPCSENDADDR	R/W	R	IPCRECVADDR
IPCSENDDATA	R/W	R	IPCRECVDATA
IPCREMOTEREPLY	R	R/W	IPCLOCALREPLY

3.2 Interprocessor Communication

Il modulo IPC consente la comunicazione tra le due CPU, e le principali funzioni sono illustrate in Fig 3.9: queste risultano indipendenti tra CPU1 e CPU2. Sono presenti due blocchi RAM 2kB dedicati, e ogni microprocessore ha accesso a lettura/scrittura su uno e sola lettura nell'altro.

Ci sono 32 segnali *IPC event* in ciascuna direzione, che possono essere utilizzati per impostare *flag* e fino a quattro *interrupt* IPC0-3 sulla CPU remota.

Ogni CPU ha otto *registri* dedicati, quattro per inviare messaggi e gli altri per riceverli; i permessi di lettura/scrittura (R/W) e i nomi dei registri sono riportati in Tab 3.1.

3.2.1 Esempio uso IPC

Le due CPU hanno la possibilità di processare due programmi differenti ed avere uno scambio di informazioni attraverso l'IPC. Un primo test preliminare prevede il TOGGLE di un GPIO, al fine di verificare la frequenza operativa. L'implementazione delle righe di codice aggiuntive non è troppo laboriosa ma prevede alcuni passaggi non banali. Il primo è quello di assegnare il controllo dei GPIO alla CPU2 attraverso il corretto registro durante l'inizializzazione dell'hardware, come si vede in Code 3.7. Vengono dunque assegnati GPIO34-63 alla seconda processing unit.

Code 3.7. tx_Hardware_Interface.c

```

39  int Initialize_the_Microprocessor_Peripherals(){
    ...
235  // Give CPU2 Control of GPIO34-63
236  GpioCtrlRegs.GPMUX1.bit.GPIO34=0; //red led
237  GpioCtrlRegs.GPBDIR.bit.GPIO34=1; //output
238  GpioDataRegs.GPBCLEAR.bit.GPIO34=1;
239  GpioCtrlRegs.GPBCSEL1.bit.GPIO34=2; //CPU2
240
241  GpioCtrlRegs.GPMUX2.bit.GPIO63=0;

```

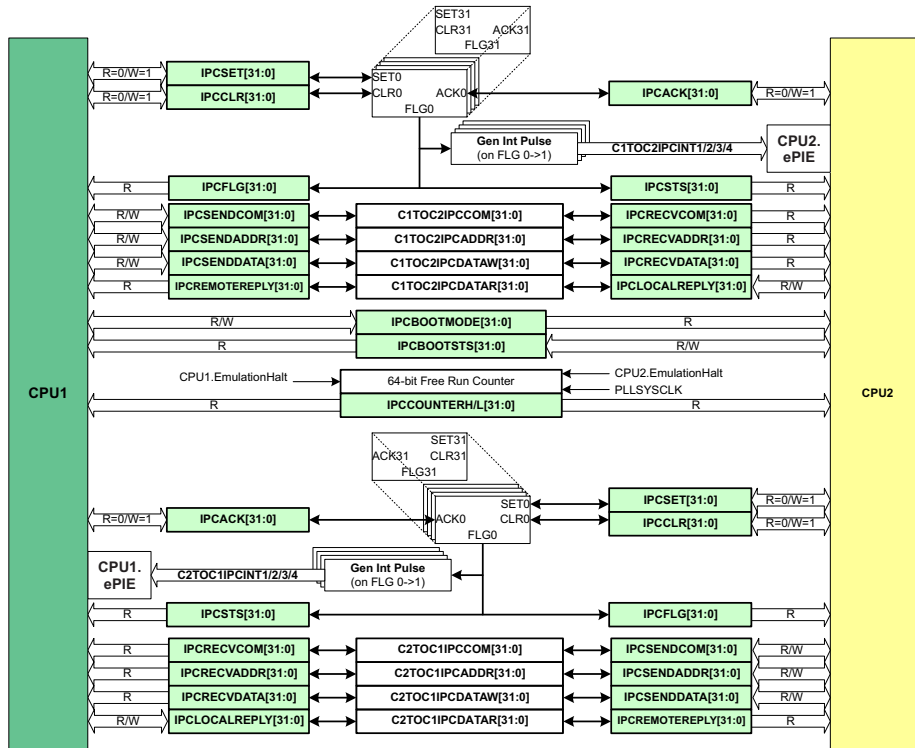


Figura 3.9. Architettura IPC

```

242 | GpioCtrlRegs.GPBDIR.bit.GPIO63=1; //output
243 | GpioDataRegs.GPBCLEAR.bit.GPIO63=1;
244 | GpioCtrlRegs.GPBCSEL4.bit.GPIO63=2; //CPU2
245 | ...}

```

Si procede poi con l'assegnazione dei flag attraverso gli *IPC event* visti in precedenza. Nel caso preso in esame, è stato scelto arbitrariamente *IPC10*: in Code 3.8 è riportato come all'interno dell'interrupt dell'ADC (ogni quattro eventi di PWM) attraverso il comando *IPCSET* viene posto a 1 lo status dell'evento *IPC*.

Code 3.8. tx_Interrupt_Service_Routines.c (CPU1)

```

39 | interrupt void ADC_isr(){
... |
238 | // TEST IPC
239 | if(IpcRegs.IPCSTS.bit.IPC10 == 0){
240 |     IpcRegs.IPCSET.bit.IPC10 = 1;
241 | }...
}

```

L'ultimo passaggio quindi è l'effettiva imposizione del *TOGGLE* nel *main* della CPU2 ogni volta che si verifica l'evento *IPC10*. È fondamentale pro-

cedere poi con la cancellazione del flag con i successivi comandi presenti in Code 3.9.

Code 3.9. main.c (CPU2)

```

104 while(1){
105     if(IpcRegs.IPCSTS.bit.IPC10 == 1){
106         EALLOW;
107         GpioDataRegs.GPBTOGGLE.bit.GPIO63 = 1;
108         EDIS;
109         IpcRegs.IPCACK.bit.IPC10 = 1;
110         IpcRegs.IPCCLR.bit.IPC10 = 1;
111     }
112     ...}

```

La verifica del corretto TOGGLE GPIO63 è riportata in Fig 3.10, dove si può notare la frequenza di 21,25kHz (bisogna guardare il "semiperiodo").

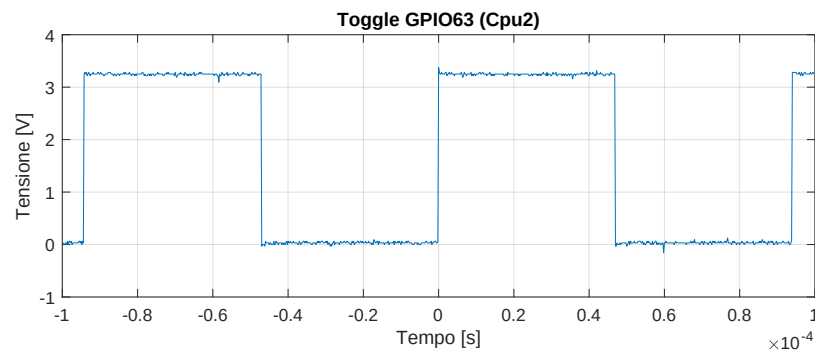
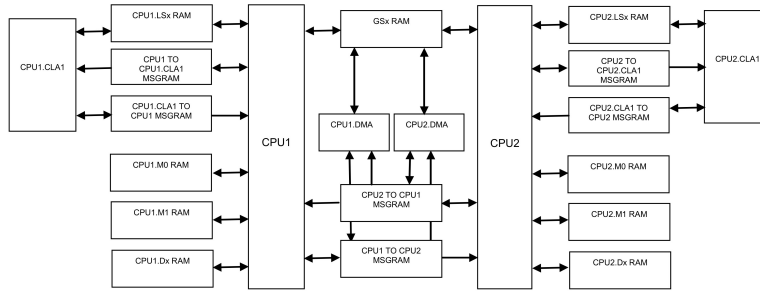


Figura 3.10. IPC uso toggle GPIO63

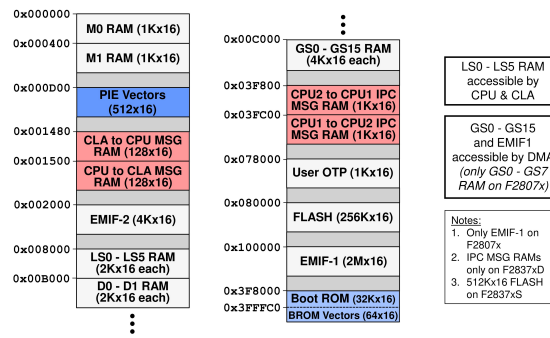
3.2.2 Variabile in GS0 RAM condivisa

La possibilità di sfruttare due CPU con programmi differenti potrebbe richiedere la capacità di gestire variabili comuni, che devono perciò essere disponibili ad entrambe le processing unit. A questo scopo nel microcontrollore sono presenti blocchi di memoria condivisa. Come si può vedere in Fig 3.11a, l'*architettura della memoria* prevede della *Global Shared GSx RAM* disponibile per le due CPU. La lettura dei dati è sempre consentita, ma la scrittura da parte di un processore deve essere preceduta da un comando nel GSx Master Selection register.

È necessario capire quali siano gli indirizzi di memoria nei quali poter operare in modo da sfruttare entrambi i core in modo intercambiabile. In Fig 3.11b ripresa dal capitolo 3. *Memory* [3] è dunque possibile individuarli.



(a) Architettura memoria



(b) Indirizzi memoria

Figura 3.11. Global Shared RAM

Usando CCS è utile ispezionare il file link 2837xD_RAM_lnk_cpu1.cmd nel quale è possibile individuare l'allocazione di memoria; parte del codice è riportata in Code 3.10 per evidenziarlo al meglio.

Code 3.10. 2837xD_RAM_lnk_cpu1.cmd

```

1  MEMORY{
    ...
26  RAMGS0      : origin = 0x00C000, length = 0x001000
27  RAMGS1      : origin = 0x00D000, length = 0x001000
28  RAMGS2      : origin = 0x00E000, length = 0x001000
    ...}
59  SECTIONS{
    ...
85  ramgs0      : > RAMGS0,      PAGE = 1
86  ramgs1      : > RAMGS1,      PAGE = 1
87  ramgs2      : > RAMGS2,      PAGE = 1
88  ...}

```

Definiamo quindi una variabile `float` che si trovi nella giusta posizione di memoria, in Code 3.11.

Code 3.11. Define_Common_Variables.c (CPU1)

```

53  float  Parameters[NUM_OF_PARAMETERS];
54  #pragma DATA_SECTION(Parameters, "ramgs0");

```

55 | ...

Nel prossimo paragrafo l'obiettivo sarà quello di verificare il corretto funzionamento del modulo SCI e la comunicazione seriale tramite protocollo RS232. Le variabili che si potranno inviare e scrivere dovranno essere disponibili da entrambe le CPU.

3.3 Connessione RS232

Il protocollo RS232 è utilizzato per scambiare informazioni tramite modulo SCI presente nel microcontrollore. Il pinout del collegamento si vede in Fig 3.12, ma non tutti i segnali vengono utilizzati: i principali sono infatti i TX e RX da parte del PC, e il GND. Per come è stata pensata la scheda di controllo è necessario un piccolo aggiustamento dei pin secondo le saldature eseguite in Fig 3.13. La comunicazione è effettuata con un adattatore da seriale a USB per l'interfaccia con il PC.

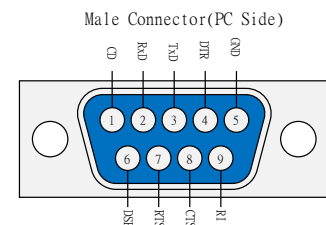
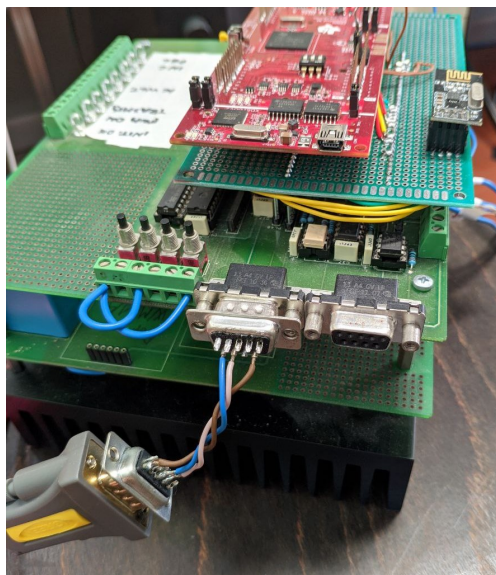
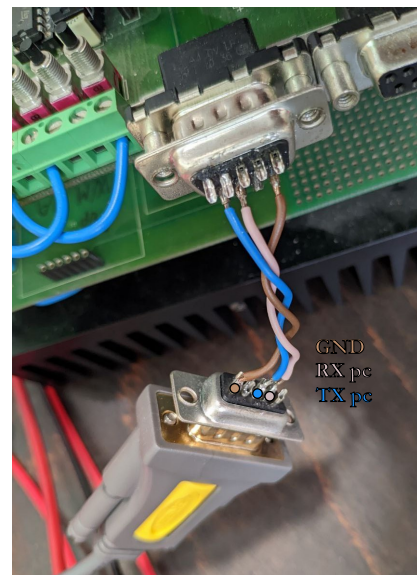


Figura 3.12



(a) Connettore scheda



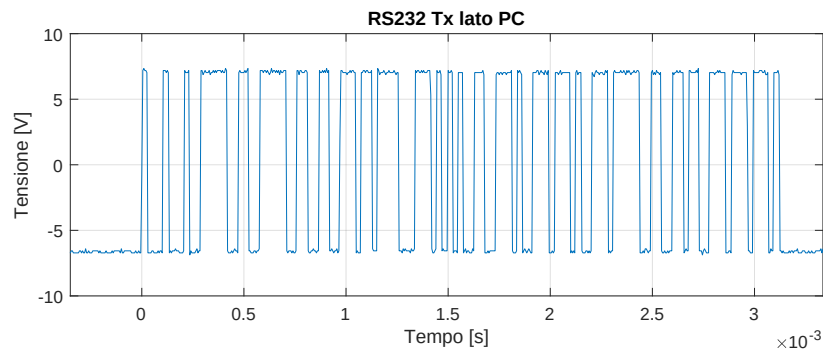
(b) Pinout

Figura 3.13. RS232 connessione al pc

Tabella 3.2. Codifica messaggi

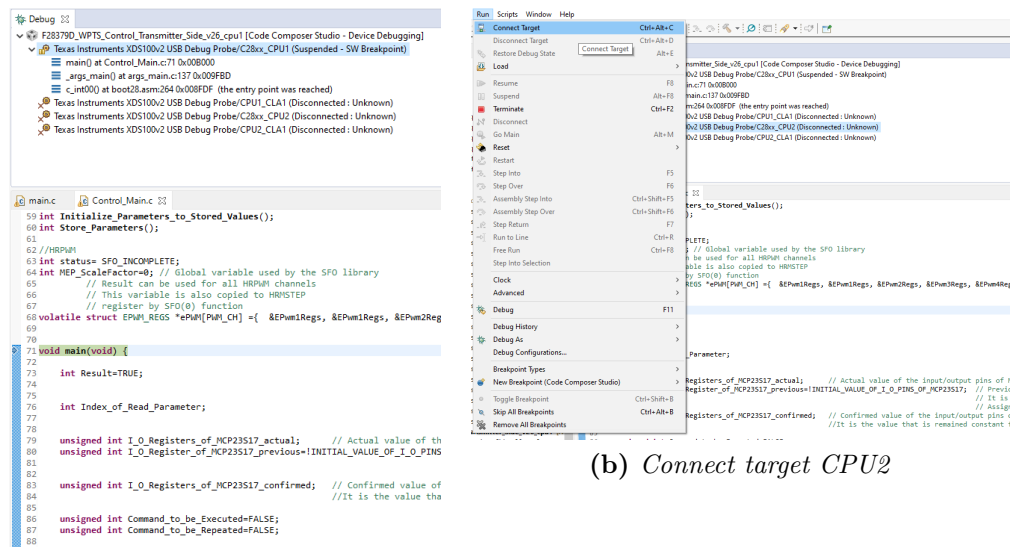
Carattere	Comando	Operazione
r	Read	Lettura parametro
w	Write	Scrittura parametro
q	Query	Lettura variabile
s	Set	Scrittura variabile

Il messaggio di prova inviato lato PC e ricevuto dal microcontrollore in Fig 3.16 illustra un esempio di come viene infine costruito il segnale TX.

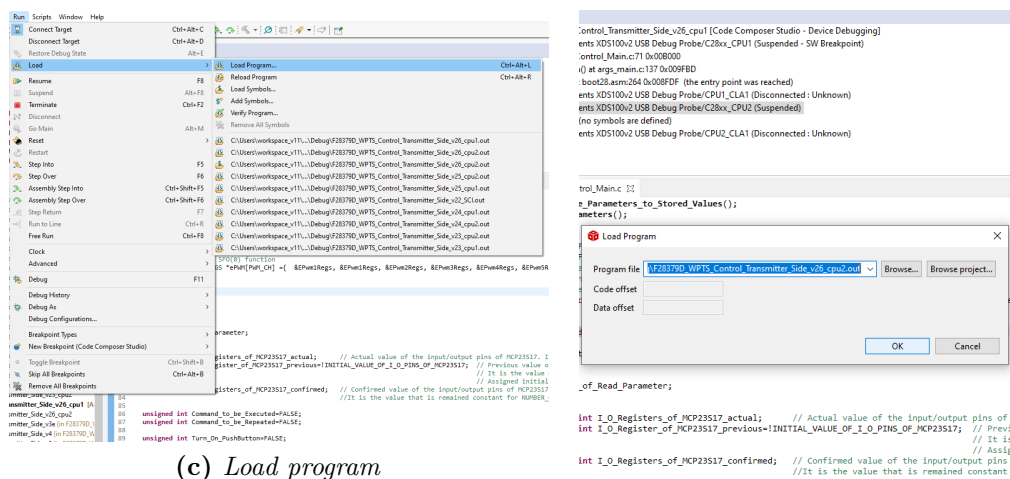
**Figura 3.16.** RS232 invio (da pc) messaggio: w00F854402D

3.4 CCS debug software dual core

Nel microcontrollore F28379D è possibile avviare due programmi differenti ciascuno per ogni core. La procedura è indicata nella *Firmware Development package Users Guide.pdf* [9] nel capitolo *2.3 Debugging Dual Core Applications* ed è riassunta di seguito. Dopo aver eseguito la build del progetto, è necessario avviare il debug del programma; la CPU1 risulta già connessa e il file .out già caricato, se correttamente impostato il file relativo al *targetConfigs*, come illustrato in Fig 3.17a. Il passaggio successivo prevede la connessione del target per la CPU2 tramite schermata in Fig 3.17b; infine bisogna procedere con il *load program* (Fig 3.17c) e selezionare il corretto file .out per la CPU2 (Fig 3.17d).



(b) Connect target CPU2



(d) File .out

Figura 3.17. Debug dual core

L'ultima operazione è quella di avviare i programmi, con l'accortezza di selezionare per primo quello della CPU1. Bisogna infine tenere a mente che nel software CCS si vedranno nella finestra di memoria i registri associati alla cpu selezionata nella debug window.

3.5 Scheda di interfaccia

La connessione tra il microcontrollore e la scheda di controllo è stata eseguita nella prima fase attraverso dei *jumper cables*, con lo schema dei collegamenti illustrato in Fig 1.4.

È risultato necessario tuttavia realizzare una *scheda di interfaccia*, per una migliore disposizione dei componenti sul banco di lavoro. Tale elemento è stato costruito a partire da una basetta millefori e da sottili cavi saldati a dei connettori *strip line* maschi (lato microcontrollore) e femmina (dalla parte dei pin della scheda di controllo).

Le saldature si sono rivelate numerose, ma grazie alla basetta scelta è stato possibile realizzarle sia nella parte superiore (Fig 3.19a) che in quella inferiore (Fig 3.19b), permettendo una migliore distribuzione dei cavi. È stato inoltre creato un connettore apposito per il modulo nRF24L01+, un ricetrasmittitore a 2,4 GHz che scambia informazioni con il microcontrollore tramite la SPI, permettendo la comunicazione tra la parte trasmittente del sistema con quella ricevente. Il risultato finale, visibile in Fig 3.18, vede quindi il sovrapporsi delle diverse schede utilizzate.

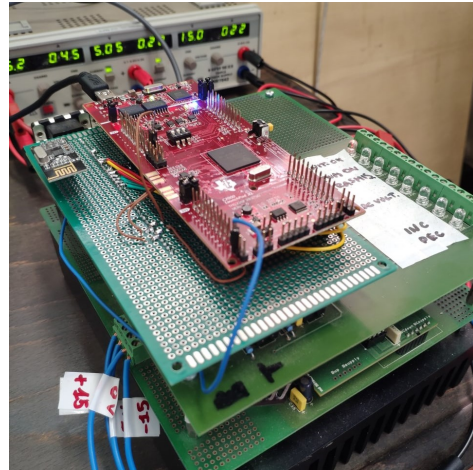
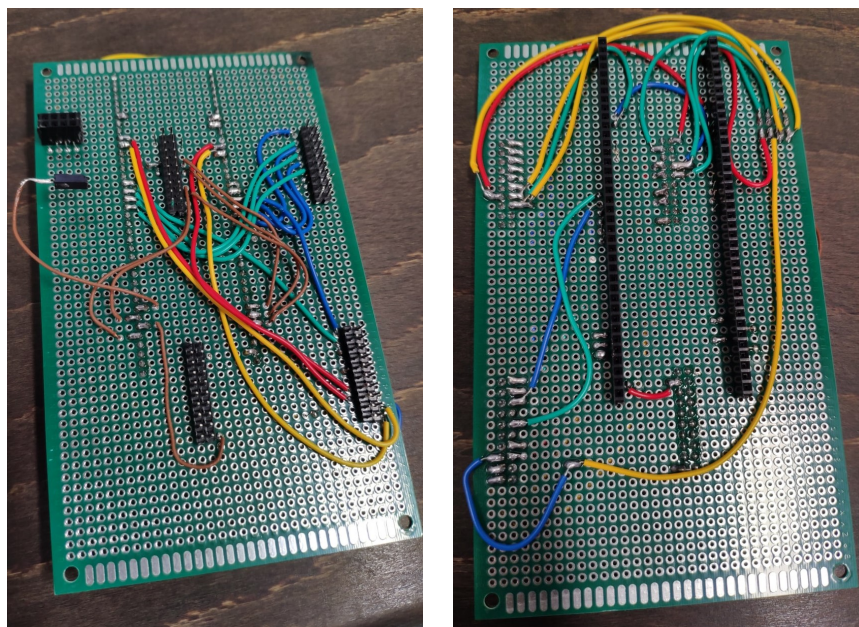


Figura 3.18. Assieme

3.6 Aggiornamenti futuri

La scheda di interfaccia è una soluzione economica e relativamente veloce da ottenere. Un'alternativa più impegnativa ma allo stesso tempo più efficace potrebbe essere quella di disegnare una pcb adatta allo scopo. In quest'ottica, è possibile pensare di implementare un connettore di espansione presente nel LAUNCHXL-F28379D, in particolare nel retro della scheda. Il connettore ad alta densità J9 (Fig 3.20a) permetterebbe l'interfaccia con più moduli e con molteplici GPIO, garantendo una maggiore aggiornabilità del sistema.

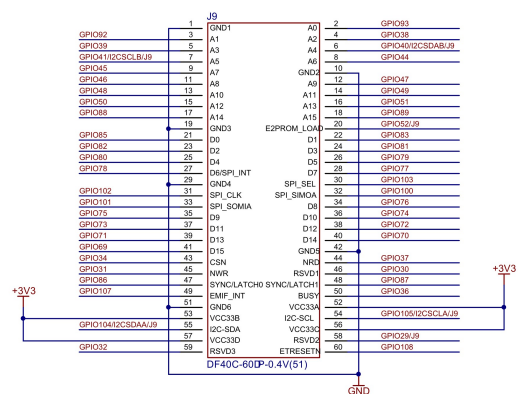
Il nome del connettore si può trovare nell'elenco dei componenti presenti [8], e si tratta di DF40C-60DP-0.4V(51) prodotto dalla Hirose Electric (Fig 3.20b).



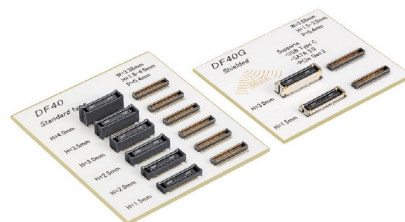
(a) Front

(b) Rear

Figura 3.19. Scheda interfaccia



(a) J9 schema



(b) DF40C-60DP-0.4V(51)

Figura 3.20. Connettore J9 (retro della scheda)

Capitolo 4

Funzionamento Dual Core

Nei paragrafi seguenti andremo ad analizzare meglio il funzionamento dual core del microcontrollore. Nella prima parte il focus si concentrerà sulla comunicazione e su come sia gestita dai due processori. La difficoltà maggiore si è riscontrata nel capire come la CPU2 sia in grado di operare sul modulo SCI, mantenendo disponibili le informazioni delle variabili anche alla CPU1.

Successivamente l'obiettivo sarà quello di verificare le *performance* del nuovo microcontrollore misurando i tempi di esecuzione, e confrontandoli con quelli del vecchio F28335. L'approccio seguito è stato assegnare un particolare valore al pin di test e misurare la distanza temporale tra due azioni consecutive.

4.1 Main CPU2

Nel precedente capitolo abbiamo già discusso del modulo di interconnessione tra i due processori. In questo paragrafo vediamo delle parti applicative in cui si fa uso massiccio dei flag e registri dedicati allo scambio di dati tra i core.

In Code 4.1 è possibile ispezionare prima di tutto l'inizializzazione della CPU2. Si utilizzano le function fornite da TI per avviare il software; si procede con l'impostazione del PIE per giungere, prima del ciclo `while`, a due passaggi fondamentali:

- imposto status IPC17 a 1, in questo modo la CPU1 può riprendere il processo di inizializzazione dopo aver atteso l'avvio della CPU2 (vedere `function Initialize_the_Hardware()`);
- inizializzazione delle periferiche, in questo caso solo SCI-A;

da notare che l'uso di moduli e GPIO deve essere preceduto da un comando della CPU1 che ne abiliti il possesso al secondo core.

Code 4.1. main.c (CPU2)

```

30 int main(void)
31 {...
32 InitSysCtrl();
33 DINT;
34 InitPieCtrl();
35 IER = 0x0000;
36 IFR = 0x0000;
37 InitPieVectTable();
38 EALLOW;
39 PieVectTable.IPC1_INT = &ipc1_ISR_CPU2;
40 EDIS;
41 PieCtrlRegs.PIEIER1.bit.INTx14 = 1;
42 IER = M_INT1;
43 EINT; // Enable Global interrupt INTM
44 ERTM; // Enable Global realtime interrupt DBGM
45
46 IpcRegs.IPCSET.bit.IPC17=1;
47 Initialize_the_Microprocessor_Peripherals();
48 EALLOW;
49 CpuSysRegs.PCLKCR7.bit.SCI_A = 1;
50 EDIS;
51
52 while(1){
53     if(IpcRegs.IPCSTS.bit.IPC10 == 1){
54         EALLOW;
55         GpioDataRegs.GPBTGGLE.bit.GPIO63 = 1;
56         EDIS;
57         IpcRegs.IPCACK.bit.IPC10 = 1;
58         IpcRegs.IPCCLR.bit.IPC10 = 1;
59     }
60 } //while
61 } //main

```

Il programma principale poi è racchiuso nel **while**, ed il suo sviluppo è semplice e banale. Siccome per il momento non ci sono particolari task da compiere, l'unica funzione è il test toggle del GPIO63, che avviene secondo l'interrupt ADC della CPU1 (Fig 3.10). Il blink del led rosso GPIO34 avviene invece nell'interrupt `ipc1_ISR_CPU2` richiamato ogni *1ms* dal Timer0 della CPU1.

4.2 RS232 management

La funzione che vuole essere assegnata alla seconda cpu è la gestione della comunicazione tramite il collegamento RS232-usb al computer. La problematica principale è relativa alla scrittura nei registri di memoria. Procediamo prima nel comprendere la struttura dello script per poi analizzare la soluzione sviluppata.

Le due funzioni che si prestano allo scopo si trovavano inizialmente nell' `if` che si verifica ogni 1ms all'interno del `while(1)` nella CPU1; si procede spostando `Manage_RS232_State_Machine(Parameters)` e la successiva `Manage_Commands_from_RS232()` nella CPU2.

La prima è riportata parzialmente in Code 4.2: i diversi `case` sono numerati e riportati con i caratteri in rosso per una maggiore comprensione grafica.

Code 4.2. RS232_management.c

```

42 | int Manage_RS232_State_Machine(float *Parameters){...

72 | while (CHAR_AVAILABLE_IN_RS232_BUFFER==TRUE){
73 |   Received_Char=CHAR_FROM_RS232_BUFFER;
   |   ...
77 |   switch (State_of_the_RS232_State_Machine){

78 |     (1)case RS232_STATE_MACHINE_WAITING_FOR_COMMAND:

79 |         RS232_Command=Received_Char;
80 |         switch (RS232_Command){
   |         case ('?'):
   |             RS232_Generate_Heart_Beat(RS232_COMMAND_LOCAL_PING);

   |         ...
   |         case ('!'):
   |             RS232_Command_not_Available(RS232_Command);
   |         ...
   |         case ('q'):
   |         case ('s'):
   |         case ('Q'):
   |         case ('S'):
   |             State_of_the_RS232_State_Machine=RS232_STATE_MACHINE_WAITING_FOR_ADDRESS;

   |         break;

   |         case ('r'):
   |         case ('w'):
   |         case ('R'):
   |         case ('W'):
   |             State_of_the_RS232_State_Machine=RS232_STATE_MACHINE_WAITING_FOR_INDEX;

   |         break;
   |         ...}
   |     break;
122 | (2)case RS232_STATE_MACHINE_WAITING_FOR_ADDRESS:

   |     if(){...if(){...if(){...
   |         switch(RS232_Command){
   |         case ('q'):
   |             RS232_Generate_Response(RS232_Command,Int_Pointer);

   |         ...
   |         case ('s'):
   |         case ('S'):
   |             State_of_the_RS232_State_Machine=RS232_STATE_MACHINE_WAITING_FOR_DATA;

   |         ...
   |         case ('Q'):

```

```

        RS232_Command_not_Available(RS232_Command);

        ...}
    } } } //if
    break;
168 (3) case RS232_STATE_MACHINE_WAITING_FOR_INDEX :

        if() { ... if() { ... if() { ...
            switch(RS232_Command) {
                case ('r'):
                    RS232_Generate_Response(RS232_Command, Int_Pointer);

                    ...
                case ('w'):
                case ('W'):
                    State_of_the_RS232_State_Machine=RS232_STATE_MACHINE_WAITING_FOR_DATA;

                    ...
                case ('R'):
                    RS232_Command_not_Available(RS232_Command);

                    ...}
            } } } //if
            break;
215 (4) case RS232_STATE_MACHINE_WAITING_FOR_DATA :

216     if() { ... if() { ... if() { ...
217         switch(RS232_Command) {
218             case ('w'):
219                 Int_Pointer=(unsigned int*)(Parameters+RS232_Index);

220                 /******//
221                 IpcRegs.IPCSENDADDR=(Uint32)Int_Pointer; //toCPU1

222                 IpcRegs.IPCSENDATA=RS232_Data;
223                 IpcRegs.IPCSET.bit.IPC1=1; //toCPU1
224                 while(IpcRegs.IPCSTS.bit.IPC18==0) {};

225                 IpcRegs.IPCACK.bit.IPC18=1;
226                 /******//
227                 *Int_Pointer=(RS232_Data>>16);
228                 Int_Pointer++;
229                 *Int_Pointer=(RS232_Data&0x0000FFFF);

230                 RS232_Generate_Response(RS232_Command, --Int_Pointer);

231                 State_of_the_RS232_State_Machine=RS232_STATE_MACHINE_WAITING_FOR_COMMAND;

232                 break;
                case ('s'): //uguale a 'w'
                    ...
                case ('S'):
                    RS232_Command_not_Available(RS232_Command);

                    ...}
            } } } //if
            break;
288 (5) case RS232...MMSB :
        ...
340 } } ...}

```

Quando un carattere è disponibile nel buffer (si è inviato quindi un coman-

do dal computer) la variabile `CHAR_AVAILABLE_IN_RS232_BUFFER` permette di entrare nel ciclo `while`. Il carattere ricevuto `CHAR_FROM_RS232_BUFFER` passa quindi ad una serie di `switch` in cascata. Non tutte i `case` sono definiti, i principali sono stati elencati nella Tab 3.2 durante la spiegazione del software *Termite*.

La struttura del messaggio inviato dal PC contiene nel primo carattere il tipo di comando; nel caso di lettura segue l'indirizzo; se si deve scrivere, viene aggiunto il dato. I principali `case` si dividono quindi:

1. `WAITING_FOR_COMMAND` attesa del comando: '?' e '!' il micro risponde con un messaggio predefinito; 'q' e 's' rimandano all'indirizzo; 'r' e 'w' aspettano l'indice;
2. `WAITING_FOR_ADDRESS` con 'q' la scheda spedisce il contenuto della memoria all'indirizzo ricevuto; con 's' aspetta i dati;
3. `WAITING_FOR_INDEX` con 'r' la scheda risponde con il dato a cui punta l'indice; con 's' aspetta i dati;
4. `WAITING_FOR_DATA` vede l'utilizzo dei registri IPC per inviare il dato alla CPU1, adibita alla scrittura; attraverso lo *status* del IPC18 il programma attenda l'avvenuta operazione.

La funzione `Manage_RS232_State_Machine` è richiamata all'interno dell'interrupt `ipc1_ISR_CPU2` ogni 1ms, grazie al Timer0 della CPU1. Il passo successivo è dunque quello di comprendere meglio i registri IPC coinvolti nella comunicazione tra i due core.

4.2.1 Registri IPC

La struttura del modulo *Interprocessor Communication* era già stata approfondita in 3.9, nell'attuale paragrafo vediamo l'applicazione dei registri dedicati.

È possibile visualizzare le IPC impostate (set), le status (sts) ed infine le interrupt service routine (isr) in Tab 4.1. Mancano tuttavia i registri per l'invio di messaggi diretti, già elencati in Tab 3.1, ed utilizzati in `WAITING_FOR_DATA`.

La Tabella è divisa a metà per distinguere i flag delle due cpu; gli usi principali sono:

SET si usa come trigger per avviare una funzione impostata sull'altro core; possiamo individuare ad esempio l'IPC17 dopo l'inizializzazione della CPU2, che permette alla CPU1 di uscire dall'attesa dopo l'avvio;

Tabella 4.1. Registri IPC nelle due CPU

	CPU 1		CPU 2	
SET	IPC10	in ADC_isr	IPC17	inizializzazione
	IPC1	in Timer0	IPC4-9	commands RS232
	IPC18	in ipc1_ISR_CPU1	IPC1	in RS232_management.c
			IPC0	in ipc1_ISR_CPU2
STS	IPC17	inizializzazione	IPC10	toggle GPIO63
	IPC4-9	commands RS232	IPC18	in RS232_management.c
ISR	IPC0	ogni 1ms	IPC1	ogni 1ms
	IPC1	gestisce dati RS232		

STS attraverso un `if` si vede lo stato dell'IPCx e nel caso in cui corrisponda ad 1 esegue un'azione;

ISR quando impostato (set) una delle prime quattro IPC(0-3), avvia una *interrupt service routine*; per entrambe le cpu sono presenti delle isr legate al timer0 ripetute ogni 1ms.

Il funzionamento dual core per *RS232 management* vede l'impiego in particolare dell'IPC18 per la scrittura dei dati e delle IPC4-9 per i comandi gestiti nella funzione `Manage_Commands_from_RS232`.

4.2.2 Esempio scambio dati

Nel capitolo precedente è stata presentata la GSRAM (global shared ram) grazie alla quale è stato possibile rendere disponibile la variabile `Parameters` ad entrambe le cpu.

Lo scambio di dati che si è in grado di instaurare tra un PC e la scheda di controllo è comprensibile nelle due figure sottostanti. Nell'immagine sinistra 4.1a si individua *memory browser*, uno strumento offerto dal debug di CCS per poter leggere i registri di memoria contenuti nel micro. Cercando nella barra superiore `&Parameters` si giunge all'indirizzo di memoria `0x0000C000`; impostando il formato di dato *32-Bit IEEE Floating Point* ne si vedono i valori nel modo corretto. Nell'immagine 4.1b sono riportati i comandi impartiti. Dobbiamo prenderci un po' di tempo per capirli appieno.

Alcuni esempi di lettura e scrittura sono spiegati meglio di seguito:

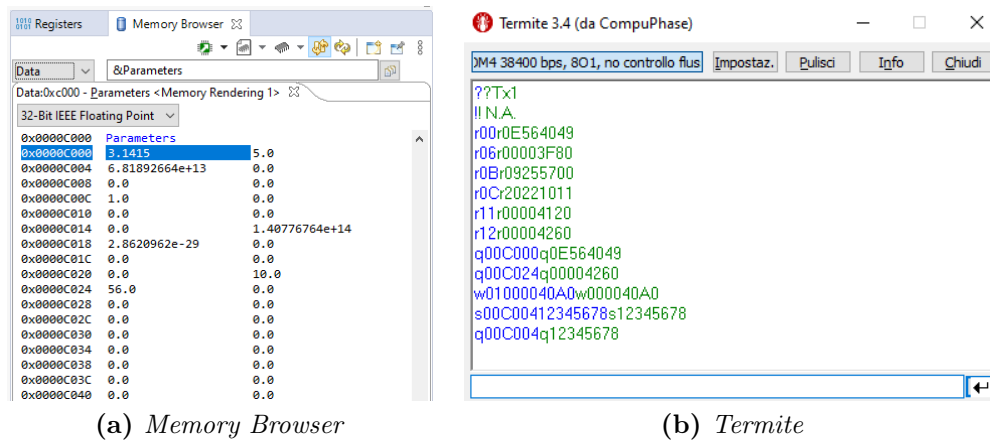


Figura 4.1. Esempio lettura e scrittura

- **r0Br092255700** richiama lettura del parametro 0B, ovvero l'undicesimo in Fig 4.1a (il primo parametro è in posizione zero); la risposta, in verde, è 0x092255700 che corrisponde a $1.40776763818e + 14$;
- **q00C024** richiama lettura dell'indirizzo 00C024; il sistema risponde con **q00004260** cioè 56 se convertito nel formato IEEE754 [16];
- **w01000040A0** richiama scrittura del parametro 01 con il dato 0x000040A0; il processore scrive quindi 5;
- **s00C00412345678** richiama scrittura nell'indirizzo 00C004 dell'informazione 0x12345678; nel memory browser è possibile quindi vedere il valore $6.81892663788e + 13$.

La capacità di scrivere e leggere i valori della variabile **Parameters** è stata quindi resa disponibile alla CPU2 per alleggerire il carico sul primo core.

4.3 Modalità standalone

Le operazioni del microcontrollore finora viste sono sempre state eseguite in RAM, i registri verificati tramite il *debug* del programma CCS. Il problema di questo funzionamento si palesa ovviamente nel momento in cui si toglie alimentazione alla scheda, che perde tutte le informazioni al suo interno. Per questo motivo vediamo ora come avviare il programma dalla memoria *flash*.

Un programma si può eseguire dalla flash cambiando il file che descrive l'architettura della memoria usato dal linker `2837xD_FLASH_lnk_cpu1.cmd`:

in questo modo il programma permane anche dopo lo spegnimento del micro. Bisogna tuttavia seguire delle ulteriori accortezze, dovute al fatto che il TMS320F28379D ha due core ed entrambi devono essere inizializzati.

Seguendo gli esempi della libreria C2000, è possibile individuare la procedura corretta. Per prima cosa si deve entrare nelle proprietà del progetto CPU1, Properties ▶ Build ▶ C2000Compiler ▶ PredefinedSymbols e definire `_FLASH` e `_STANDALONE`. Si prosegue infine all'aggiunta di codice in `Initialize_the_Hardware()` visibile in Code 4.3.

Code 4.3. Hardware_Interface.c (CPU1)

```

43 int Initialize_the_Hardware(){
44     int Result=TRUE;
45     InitSysCtrl(); //include DisableDog
46
47     #ifdef _STANDALONE
48     #ifdef _FLASH
49     IPCBootCPU2(C1C2_BROM_BOOTMODE_BOOT_FROM_FLASH);
50
51     #else
52     IPCBootCPU2(C1C2_BROM_BOOTMODE_BOOT_FROM_RAM);
53
54     #endif
55     #endif
56
57     #ifdef _FLASH
58     InitFlash();
59     #endif
60     ...}

```

Allo stesso modo per la CPU2, nei *simboli predefiniti* si definisce `_FLASH` e si aggiunge il file `2837xD_FLASH_lnk_cpu2.cmd`. La porzione di codice aggiuntiva è riportata in Code 4.4.

Code 4.4. main.c (CPU2)

```

42 int main(void){
43     //Copy time critical code and Flash setup code to RAM
44
45     #ifdef _FLASH
46     memcpy(&RamfuncsRunStart,&RamfuncsLoadStart,(size_t)&RamfuncsLoadSize);
47
48     #endif
49
50     InitSysCtrl();
51
52     #ifdef _FLASH
53     InitFlash();
54     #endif
55     ...}

```

Una volta configurato il tutto, si caricano i programmi nel microcontrollore. Chiudendo CCS e riavviando l'alimentazione al dispositivo, è possibile vedere il corretto funzionamento del codice attraverso i due led GPIO31-34 che assicurano il giusto scambio di informazioni tramite IPC tra i due core.

4.4 Prestazioni

Valutiamo infine le prestazioni del sistema: per misurare i tempi di esecuzione si percorrono due approcci. Il primo è quello di impostare un **TOGGLE** all'inizio del ciclo `while(1)` nel main della CPU1, in questo modo lo stato del pin considerato per il test cambierà all'inizio del ciclo successivo. Il secondo prevede invece l'uso dei registri **SET** e **CLEAR** per lo stesso GPIO di test rispettivamente all'inizio e alla fine di `ADC_isr()`, consentendo la misura del tempo impiegato.

Il confronto tra i due microcontrollori della TI conclude l'analisi. Per poter paragonarli al meglio è necessario aggiungere dei cicli `for` all'interno della routine di servizio dell'interrupt adc che simulino dei calcoli effettuati dal processore; in questo modo abbiamo un'idea dell'incremento di performance.

4.4.1 While

Si procede dunque come spiegato all'impostazione di un toggle nel `while` principale, si misura il tempo di esecuzione tra un ciclo e l'altro. In Fig 4.2 sono riportate quattro misurazioni dello stesso segnale con scale temporali diverse sia per il nuovo microcontrollore F28379D, che per il vecchio F28335.

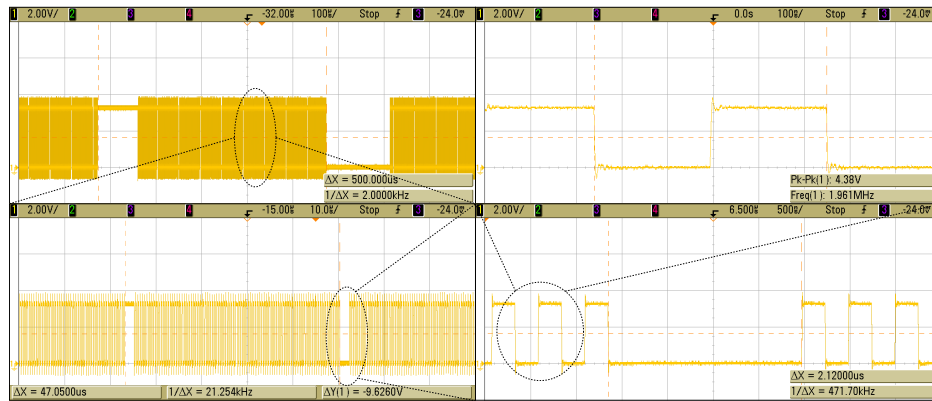
Possiamo notare come il toggle non abbia un andamento costante, ma è interrotto dai diversi *interrupt* che sono richiamati durante lo svolgimento del main. Nelle due figure a sinistra si apprezza questo fenomeno; nelle immagini inferiori si vede il ripetersi con una frequenza pari a 21,25kHz di un intervallo in cui il toggle viene sospeso dovuto all'*isr* dell'ADC. Già a questo punto è possibile apprezzare come il vecchio microcontrollore impieghi poco meno di 5μs per eseguire la isr contro i poco più dei 2μs del nuovo. Entrambe le misure sono state effettuate con pwm spenta.

Infine, nella finestra superiore destra, si misura effettivamente il tempo tra un ciclo e l'altro:

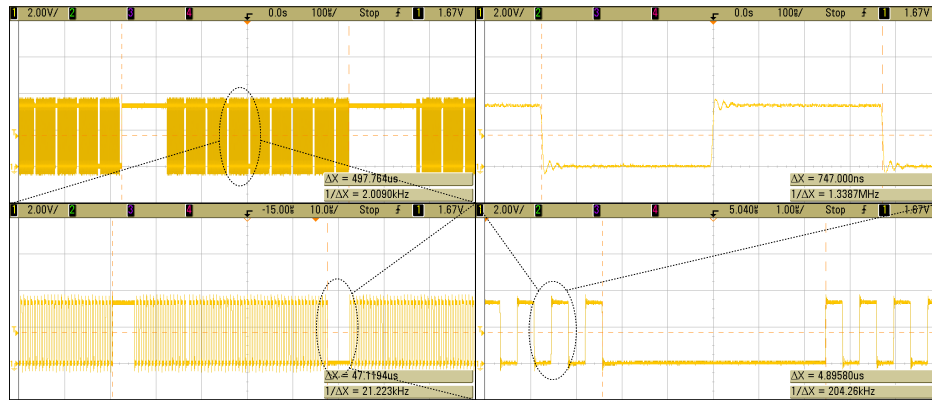
$$\frac{1}{1.961\text{MHz} \cdot 2} = 255\text{ns} \quad \text{per il nuovo F28379D}$$

$$747\text{ns}/2 = 373.5\text{ns} \quad \text{per il vecchio F28335}$$

si capisce quindi che per il ciclo `while` il nuovo micro ci impiega un $\sim 30\%$ in meno rispetto al precedente grazie al maggior clock e alle minori operazioni assegnate alla CPU1.



(a) GPIO32 nel F28379D



(b) GPIO34 nel F28335

Figura 4.2. Toggle nel ciclo while(1)

4.4.2 Interrupt ADC

La misura del tempo di esecuzione dell'interrupt service routine dell'ADC avviene ponendo i comandi SET e CLEAR del pin di test GPIO. Volendo simulare il comportamento operativo, è stato disabilitato `MANUAL_DUTY_CYCLE` e abilitato `tx_Amplitude_Modulated_Supply_Voltage()`; il funzionamento degli allarmi è rimasto disattivato per le misure riportate in Fig 4.3.

Le tempistiche di esecuzione sono divise in due per entrambi i micro, da una parte i tempi con pwm spenta e dall'altra accesa. Tale scelta serve ad evidenziare le diverse performance per le azioni svolte:

- *pwm disattivata* non si impostano i registri dei moduli; in questo caso il nuovo F28379D impiega $2.32\mu s$ contro i $4.23\mu s$ del precedente; il netto miglioramento è dovuto alla presenza di più moduli ADC che campionano simultaneamente fino a quattro segnali alla volta;

- *pwm attivata* l'incremento prestazionale è più contenuto, infatti il vecchio F28335 impiega $5.03\mu s$ contro gli attuali $4.45\mu s$; la differenza è meno marcata a causa degli ulteriori registri impostati nella isr, come IPC e soprattutto quelli per la pwm ad alta risoluzione.

Nonostante i tempi relativamente più contenuti, bisogna tenere a mente che il vero vantaggio si può avere grazie alla capacità di calcolo superiore; nel prossimo paragrafo viene valutato quest'aspetto.

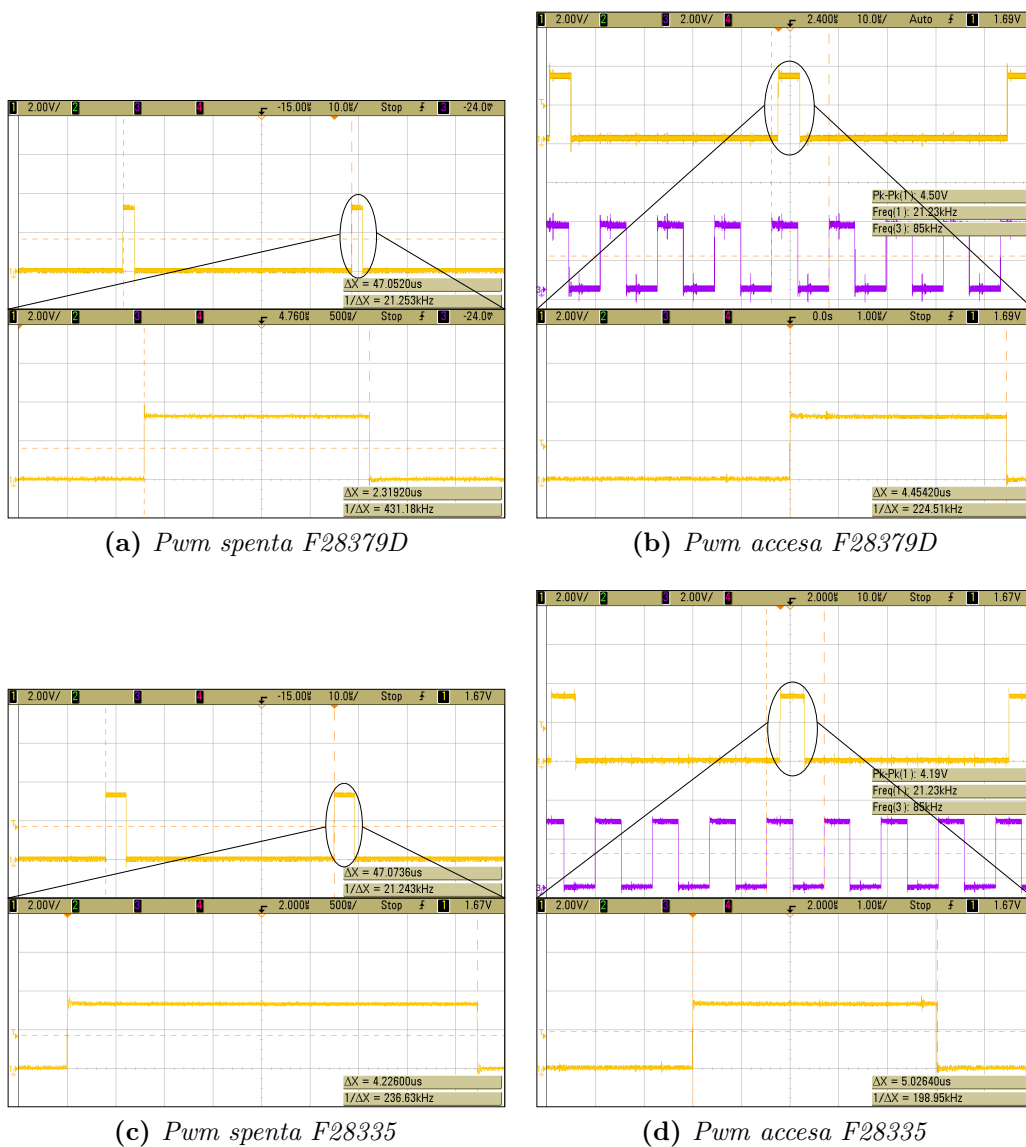


Figura 4.3. Set e Clear nell'interrupt ADC, PWM1A (viola)

Questo tipo di controllo produce delle forme ad onda quadra, in cui le uscite pwm sono sfasate tra loro di $2\pi/3$, pari a

$$\frac{11.76}{3} = 3.92\mu s$$

e sono visibili in Fig 4.4. Il nuovo micro permette l'utilizzo dei CMP ad alta risoluzione, che garantiscono una precisione maggiore.

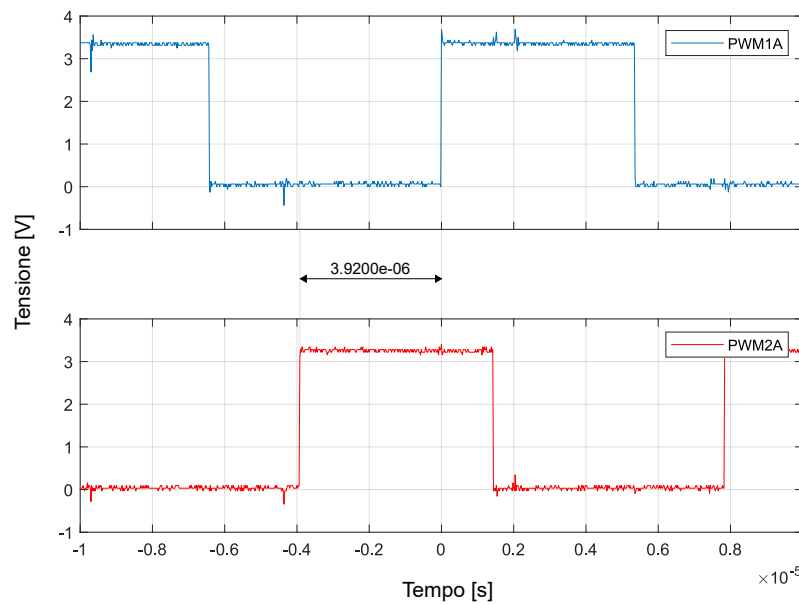


Figura 4.4. Segnali pwm sfasati di $2\pi/3$

4.4.3 Confronto microcontrollori

Dopo aver misurato i tempi per l'esecuzione delle configurazioni e del codice già scritto, è necessario confrontare i due microcontrollori simulando una serie di calcoli. Lo scopo è impegnare le due MCU mediante un numero n di cicli `for` test. Le righe di codice, alquanto banali e senza un particolare significato, sono riportate in Code 4.5 e servono solo ad occupare tempo computazionale.

Code 4.5. tx_Interrupt_Service_Routines.c

```

39 | interrupt void ADC_isr(){
    | ...
198 |     long i;
199 |     long a;

```



```

200     for(i=1; i<n; i++){
201         a=(i+9)*22;
202         a++;
203     }
204     ...}

```

La Tab 4.2 contiene i dati delle misurazioni effettuate con un numero di cicli variabile. L'attività degli allarmi all'interno della *ADC_isr* è stata ripristinata.

Tabella 4.2. Tempi impiegati ciclo `for`

Numero cicli	F28335 [μs]		F28379D [μs]		Risparmio tempo [%]
	ADC	+PWM	ADC	+PWM	
0	4,23	5,03	2,60	4,75	5,6
100	20,77	21,57	15,01	17,15	20,5
200	37,44	38,24	27,51	29,65	22,5
300	54,11	54,91	40,02	42,16	23,2
500	87,46	88,26	65,01	67,15	23,9

Il grafico in Fig 4.5 è stato realizzato riportando i risultati delle misurazioni sopra elencate. Come era logico aspettarsi, si ottengono delle rette che rappresentano l'impiego di tempo proporzionale al numero di cicli effettuati. In blu e in ciano distinguiamo il vecchio F28335, mentre il nuovo F28379D è rappresentato dal rosso e magenta: la differenza temporale tra pwm accesa e spenta rimane costante, rendendo le rette a due a due parallele.

Viene valutato quanto tempo si risparmia in percentuale rispetto al precedente micro; il calcolo

$$Risparmio\ tempo\ [\%] = \frac{t_{pwmold} - t_{pwmnew}}{t_{pwmold}} \cdot 100$$

riporta quanto effettivamente convenga la nuova scheda. L'incremento teorico, tenendo presente solo il clock, dovrebbe essere del 25% ($SYSCLK_{F28335} = 150\text{MHz}$ contro $SYSCLK_{F28379D} = 200\text{MHz}$). Tale valore non viene raggiunto a causa delle nuove operazioni impostate.

Notiamo un particolarità: il nuovo microcontrollore risulta più vantaggioso con un numero di cicli alto. Non è possibile tuttavia raggiungerne uno troppo elevato a causa del limite temporale dettato dalla isr dell'ADC. Quest'ultima lavorando a 21.25kHz impone di non superare il confine dei 47.04 μs , rappresentato da una linea tratteggiata nel grafico.

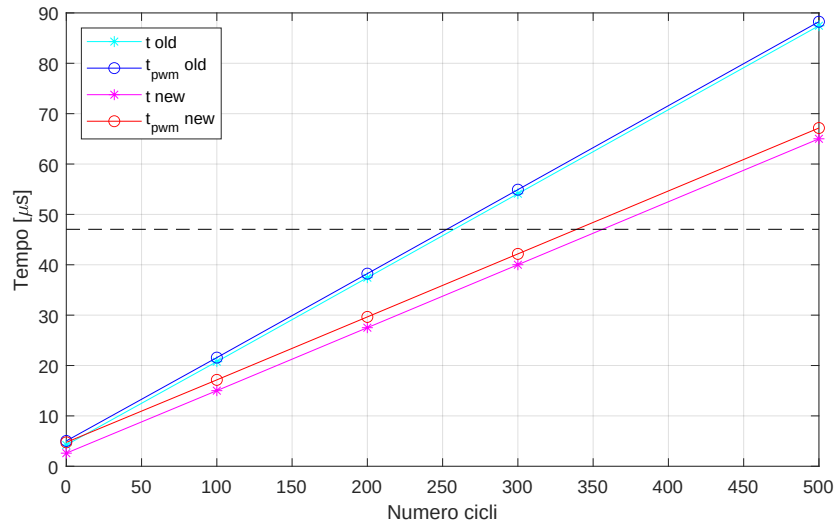


Figura 4.5. Tempi esecuzione *ADC_isr* (F28335 vs F28379D)

Se si procede oltre il micro non è in grado di gestire l'ADC ogni quattro periodi di pwm, non garantendo il comportamento ottimale del sistema. Tale aspetto si vede bene in Fig 4.6, dove il vecchio micro non è in grado di raggiungere i 300 cicli senza superare il limite; la isr avviene quindi ogni otto periodi di pwm.

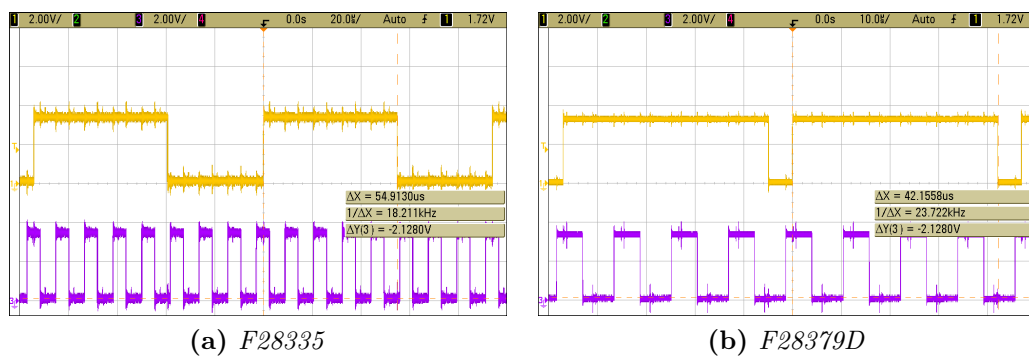


Figura 4.6. Set e Clear in *ADC_isr* (giallo), PWM1A (viola)
 Numero cicli for : 300

4.5 Collegamento al sistema completo

Come verifica conclusiva è stato collegato il microcontrollore con le schede di controllo e potenza al sistema completo, che include le bobine per l'accoppiamento e il carico. L'obiettivo è quello di vedere le forme d'onda lato trasmettente, dove si trova il MCU, e lato ricevente. È stata utilizzata una tensione di prova del bus DC nominale pari a 45V. Prima di avviare la pwm, bisogna cambiare le impostazioni relative al modulo PWM1 della scheda di potenza, che risulta non funzionante, sostituendolo con PWM3. Definendo `SWAP_TRANSMITTER_PHASES` nel file `tx_Test_Definitions.h` si effettuano due modifiche:

- `tx_Definitions.h` nella sezione *PWM gate signals definition* cambiano i registri che verranno poi impostati nella `ADC_isr`;
- `tx_Hardware_Interface.c` alla fine di *Initialize PWM*, il modulo `ePwm3` è inizializzato al posto del primo.

Infine l'algoritmo controllo è disattivato, e si definisce `MANUAL_DUTY_CYCLE` per selezionare lo phase shift attraverso i pulsanti esterni.

4.5.1 Strumenti

Gli strumenti per alimentare il sistema e per prendere le misurazioni, riportati in Fig 4.7, sono:

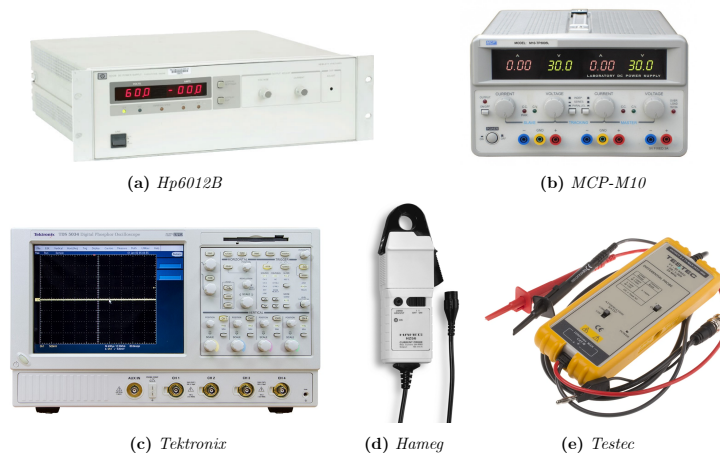


Figura 4.7. Strumenti sul banco

- (a) *Hp 6012B* DC power supply con intervallo di tensione 0-60V, di corrente 0-50A e potenza 1000W; è stato alimentato il bus DC con tensione massima di 48V;

- (b) *MCP* modello *M10-TP3003L* usato per alimentare la scheda di controllo a 15V e 5V;
- (c) *Tektronix TDS 5034* oscilloscopio al quale sono state connesse le sonde di tensione e corrente; il numero di campioni acquisiti è pari a 50000, con tempo di campionamento $4\text{e-}10\text{s}$ (Fig 4.8);
- (d) *Testec SI-9002* sonda di tensione con attenuation ratio pari a 1/20;
- (e) *Hameg HZ56* sonda di corrente, con uscita di 100mV/A.

4.5.2 Risultati

Le misurazioni sono state acquisite e poi elaborate in MATLAB; la Fig 4.8 illustra tensioni e correnti della parte trasmittente e di quella ricevente.

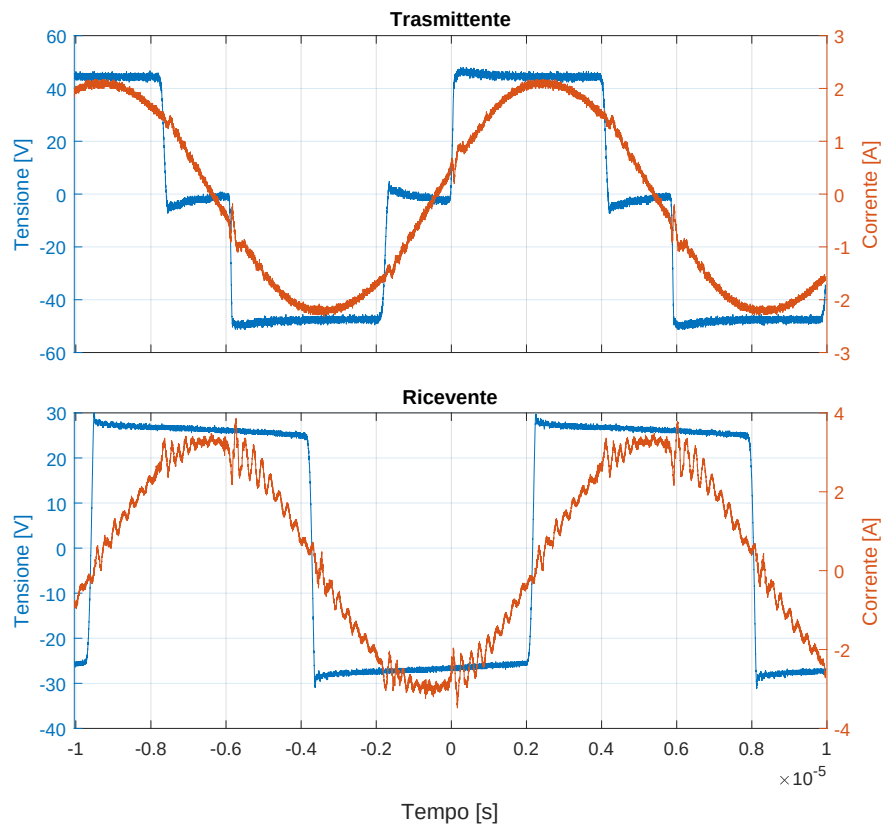


Figura 4.8. Forme d'onda sistema completo

La tensione massima utilizzata è stata 48V per alimentare il bus DC, ben al di sotto degli oltre trecento per cui è stato dimensionato il sistema. L'obiettivo tuttavia è verificare la correttezza nelle forme d'onda: con estrema soddisfazione si può apprezzare quella della tensione lato trasmittente e della corrente sinusoidale, mentre lato ricevente si nota la tensione ad onda quadra prima del ponte a diodi.

Appendice A

Schemi completi

Nella presente appendice sono riportati alcuni schemi completi delle schede di controllo e di potenza utilizzati durante la programmazione del microcontrollore. Si sono rivelati utili molteplici volte, per poter ispezionare al meglio soprattutto la scheda di controllo; non è stato raro infatti imbattersi in errori risolvibili solo grazie alla consultazione degli schemi e alla visualizzazione della pcb, non riportata. Il software usato è *DesignSpark PCB v10*.

In Fig A.1 si può vedere lo *Schema completo scheda di controllo*; tra le parti principali si notano i quattro pulsanti per attivare, disattivare e modificare il phase shift della pwm e i diversi led che indicano lo stato del convertitore. Si palesa oltretutto la presenza di moduli per la comunicazione SPI, ed infine i pin dell'intera scheda. Come già menzionato, la disposizione di questi ultimi è differente dalla configurazione del LAUNCHXL-F28379D; si è adottato quindi una scheda di interfaccia per garantire un collegamento e una gestione degli spazi migliore.

In Fig A.2 si trova invece lo *Schema completo scheda di potenza*, grazie al quale è possibile individuare i collegamenti degli switch ed altri componenti. Questa board è direttamente fissata ad un radiatore per consentire una dissipazione ottimale.

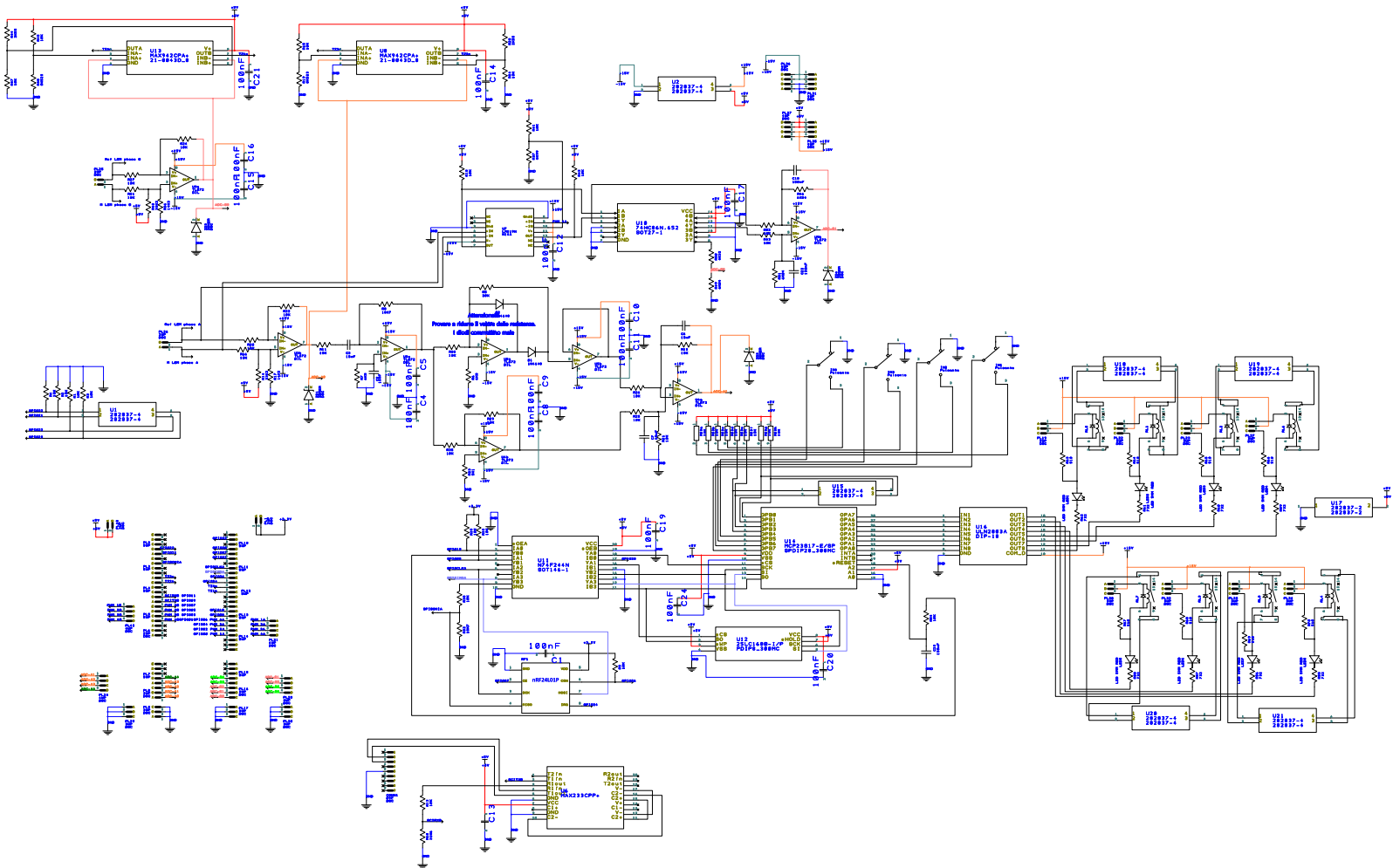


Figura A.1. Schema completo scheda di controllo

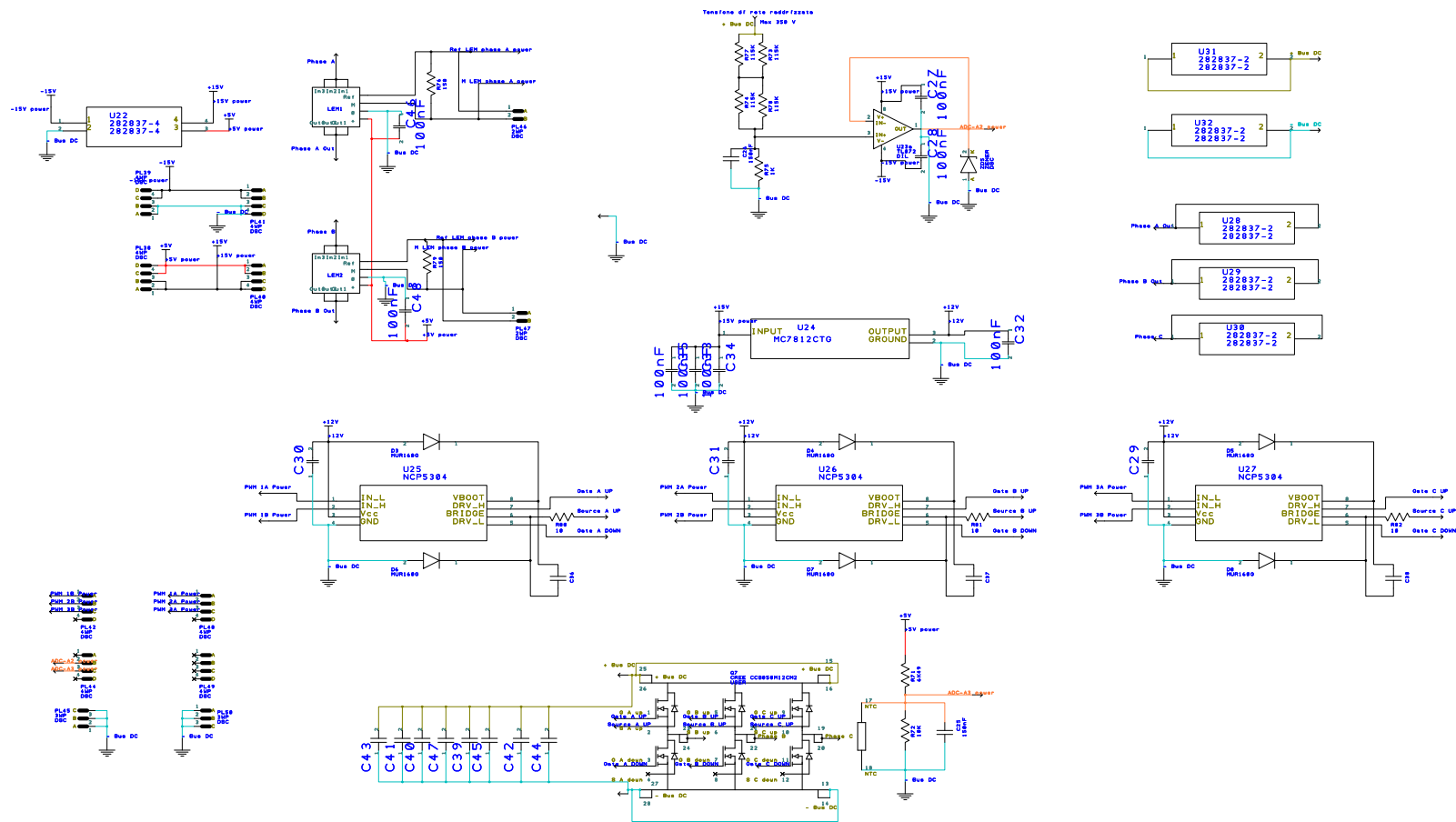


Figura A.2. Schema completo scheda di potenza

Conclusioni

In conclusione a questo lavoro di tesi è necessaria un'ultima analisi per indicare quali sono i vantaggi nell'implementare il firmware di controllo per un nuovo microcontrollore, e quali possono essere invece gli eventuali limiti. L'attività di programmazione si è svolta totalmente nel software CCS Code Composer Studio, grazie al quale è possibile effettuare il *debug* in maniera efficace: visualizzare i registri coinvolti in tempo reale è fondamentale per capire il corretto funzionamento o eventuali problemi della scheda di sviluppo LAUNCHXL-F28379D. La scelta di questo prodotto si è rivelata azzeccata grazie alla parziale compatibilità con il precedente chip TMS320F28335; non tutte le funzioni sono rimaste uguali e la maggior parte di esse è stata revisionata, ma è stata garantita una buona base di partenza.

I vantaggi della nuova MCU sono palesi ed oltre le prestazioni superiori, evidenziate nell'ultimo capitolo, sono molteplici i benefici per le implementazioni future, a partire dal numero maggiore di periferiche. I problemi tuttavia non sono mancati durante la realizzazione del nuovo codice. La trattazione teorica riguardante l'architettura dei vari moduli è affidata al manuale, che illustra il più delle volte tutti i registri necessari al funzionamento. In alcuni casi però l'ordine preciso di configurazione è meglio individuarlo all'interno degli esempi specifici delle librerie C2000. Le difficoltà si incontrano ciononostante quando si utilizzano periferiche diverse contemporaneamente; un esempio tra tutti l'impostazione di un ingresso *Crossbar* che deve essere effettuata prima dell'abilitazione del Trip-Zone. Questo tipo di informazioni si trova nelle note del manuale nella versione più aggiornata. Un ulteriore limite del sistema è stato riscontrato nel Paragrafo 3.1 relativo alla pwm ad alta risoluzione. Per le condizioni di funzionamento desiderate non è stato possibile utilizzare i registri HR del Time Base Period.

Malgrado i piccoli imprevisti incontrati lungo il percorso, è stato interessante comprendere al meglio il funzionamento interno del microcontrollore e la struttura del software di controllo. Proprio quest'ultima è stata riadattata anche in funzione della presenza di due core. Il modulo IPC è basilare per lo scambio di informazioni tra le CPU.

Il lavoro ha avuto l'obiettivo di garantire la compatibilità delle precedenti funzioni e implementare periferiche aggiuntive. Le prestazioni superiori sono state verificate e possono permettere l'esecuzione di algoritmi di controllo più complessi. Ciò che non è stato evidenziato nel Paragrafo 4.4 è l'eventuale uso contemporaneo di *interrupt service routine* nei due core per ottenere performance doppie rispetto all'attuale sistema. Un ulteriore aspetto su cui si può lavorare è la progettazione di una scheda di interfaccia dedicata, come suggerito negli Aggiornamenti futuri 3.6.

Lo sviluppo del nuovo firmware non può quindi considerarsi un punto di arrivo, bensì di partenza.

Bibliografia

- [1] Texas Instruments, *TMS320F2837xD Dual-Core Microcontrollers Technical Reference Manual (Rev. I)* ,
disponibile a: https://www.ti.com/lit/ug/spruhm8i/spruhm8i.pdf?ts=1655383776417&ref_url=https%253A%252F%252Fe2e.ti.com%252F
(Sep. 05, 2019)
- [2] Texas Instruments, *TMS320F28379D LaunchPad Quick Start Guide (Rev. A)*,
disponibile a:
https://www.ti.com/lit/ml/sprui73a/sprui73a.pdf?ts=1655394961821&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FLAUNCHXL-F28379D
(Aug. 11, 2017)
- [3] Texas Instruments, *The TMS320F2837xD Architecture: Achieving a New Level of High Performance* ,
disponibile a: https://www.ti.com/lit/an/sprt720/sprt720.pdf?ts=1655382911902&ref_url=https%253A%252F%252Fwww.google.com%252F
(Feb. 11, 2016)
- [4] Texas Instruments, *The Essential Guide for Developing With C2000 Real-Time Microcontrollers (Rev. D)*,
disponibile a:
https://www.ti.com/lit/an/spracn0d/spracn0d.pdf?ts=1655449464559&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FLAUNCHXL-F28379D
(Sep. 14, 2021)
- [5] Texas Instruments, *TMS320F2837xD Dual-Core Microcontrollers datasheet (Rev. O)* ,
disponibile a:
https://www.ti.com/lit/ds/sprs880o/sprs880o.pdf?ts=1655403811680&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FTMDSDOCK28379D%253FHQS%253Ddis-dk-null-digikeymode-dsf-pf-null-wwe%2526DCM%253Dyes%2526distId%253D10

- (Jan. 18, 2021)
- [6] Texas Instruments, *C2000™ Unique Device Number (Rev. B)*,
disponibile a:
https://www.ti.com/lit/an/spracd0b/spracd0b.pdf?ts=1655449466253&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FLAUNCHXL-F28379D
(Sep. 17, 2020)
- [7] Texas Instruments, *Piccolo™ C2000™ MCUs enable the next generation of low-cost, dual-axis servo dri (Rev. A)*,
disponibile a:
<https://www.ti.com/lit/wp/sway022a/sway022a.pdf?ts=1655388036707>
(Jul. 13, 2020)
- [8] Texas Instruments, *LAUNCHXL-F28379D overview user's guide (Rev. C)* ,
disponibile a: https://www.ti.com/lit/ug/sprui77c/sprui77c.pdf?ts=1655401835321&ref_url=https%253A%252F%252Fwww.google.com%252F
(Mar. 28, 2019)
- [9] Texas Instruments, *F2837xD Firmware Development Package* ,
disponibile a: https://dev.ti.com/tirex/explore/node?node=AAbb6ca61Sq9Tf36VqdbuA__coGQ502__LATEST
(visitato il 17 giugno 2022)
- [10] Mario Di Raimondo, *Tricky C*,
disponibile a: https://diraimondo.dmi.unict.it/wp-content/uploads/classes/so/mirror-stuff/Tricky_C.pdf
(visitato il 17 giugno 2022)
- [11] IEEE; Autori: Giuseppe Buja, Manuele Bertoluzzo, Kishore Naik Mude; *Design and Experimentation of WPT Charger for Electric City-Car* ,
disponibile a: <https://ieeexplore.ieee.org/document/7155552>
(Date of Publication: 13 July 2015)
- [12] Cesare Franzin; Università di Padova, Tesi di Laurea Magistrale in Ingegneria dell'Energia Elettrica,
Analisi delle potenzialità di un microcontrollore Dual-Core,
(Anno accademico 2020/2021)
- [13] Software Texas Instruments, *Code Composer Studio Downloads* ,
disponibile a:
https://software-dl.ti.com/ccs/esd/documents/ccs_downloads.html

(Version: 11.1.0, Release date: 21 Dec 2021)

- [14] Software Texas Instruments, *C2000Ware for C2000 MCUs* ,
disponibile a: <https://www.ti.com/tool/C2000WARE>
(Version: 4.01.00.00, Release date: 03 Mar 2022)
- [15] Hameg, *Triple Power Supply HM7042* ,
disponibile a:
https://cdn.rohde-schwarz.com/hameg-archive/data_en_HM7042-3.pdf
(visitato il 17 giugno 2022)
- [16] Tool, *IEEE-754 Floating Point Converter* ,
disponibile a: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>
(visitato il 17 giugno 2022)