UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

TESI DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

Sviluppo del firmware per il controllo di un'automazione per portoni su microcontrollore a 32 bit

Relatore: Prof. Michele Moro

Laureando: Enrico Tomasella
Matr. 521952

AA 2010/2011

Indice

Elenco delle tabelle

Elenco delle figure

Sommario

Il lavoro si è svolto nell'ufficio elettronico all'interno di una ditta che opera nel campo del'*Home Automation* e ha il compito di sviluppare le schede elettriche di controllo dei vari prodotti. L'obbiettivo è quello di arrivare ad una gestione aziendale dei progetti più veloce, con lo scopo di ridurre il *time to market* dei nuovi prodotti e agevolarne il mantenimento.

La parte laboriosa consiste nella scrittura del firmware in *assembler* e nel suo collaudo che sono semplificabili utilizzando *ambienti di sviluppo* a più alto livello e che introducano strumenti di diagnostica più potenti e versatili. Queste potenzialità non combaciavano fino a poco tempo fa con le limitate risorse hardware utilizzate in questo settore ma la recente diffusione dei dispositivi più prestanti ne hanno fatto crollare il prezzo aprendoci a queste nuove possibilità.

Si sono ricercati i componenti hardware e gli strumenti software più adatti ai nostri scopi e si è partecipato alla progettazione di un nuovo prodotto realizzando una architettura modulare che permette il riutilizzo del codice e la sua portabilità.

Capitolo 1

Introduzione

In ambito industriale si ha la necessità di azionare portoni destinati a effettuare un elevato numero di manovre in maniera molto veloce e affidabile. A differenza delle alternative domestiche, dove si utilizzano prevalentemente motori in corrente continua con controllo in PWM, è necessario sfruttare le caratteristiche dei motori in corrente alternata pilotati tramite inverter trifase.

Il lavoro consiste nel realizzare la centrale di un'automazione per portoni industriali verticali. Una centrale è una scheda elettrica di controllo alla quale sono collegati vari ingressi (come ad esempio fotocellule, ricevitore radio, encoder, finecorsa, pulsantiere, transponder) e degli attuatori (come lampeggianti, motore, semaforo). Ci sono inoltre una serie di funzionalità aggiuntive utili alla sua configurazione e manutenzione quali pulsanti e led.

Il contesto affidabile richiede, oltre a una realizzazione meccanica adatta, anche la possibilità di eseguire una diagnostica da remoto e di controllare a distanza l'automatismo. Spesso, inoltre, per movimentare la stessa struttura è necessaria la collaborazione tra due motori diversi (master e slave). Entrambe queste necessità sono state rese possibili utilizzando un bus di comunicazione seriale proprietario (bus T4) che colleghi ogni centrale e che è stato usato poi anche per estendere le funzioni di configurazione.

Parte del codice e del protocollo seriale era già disponibile ma si è lavorato per rendere il progetto portabile e il più possibile indipendente dalla piattaforma hardware scegliendo una struttura modulare e utilizzando microcontrollori prestanti.

In questa particolare applicazione il motore verrà collocato distante dalla centrale per aumentarne l'immunità ai disturbi. Il pilotaggio diretto viene eseguito da un modulo inverter dedicato collegato via seriale al quale vengono passate le informazioni riguardanti le rampe di accelerazione e velocità e dal

quale si ricevono le quote ricavate dall'encoder di posizione assoluto e la corrente assorbita.

L'azionamento automatico di un portone richiede, oltre al controllo del dispositivo che lo mette in moto, anche un monitoraggio del suo stato, dei comandi che può continuamente ricevere e delle condizioni di sicurezza. È necessario quindi disporre di uno strumento capace di interfacciarsi con l'applicazione per leggere gli ingressi, pilotare le uscite e che abbia un modo per impostare la logica che deve seguire. Questo è il campo proprio dei microcontrollori, circuiti elettronici integrati che comprendono oltre ad una unità logica di elaborazione dati anche zone di memoria, timer, moduli di comunicazione e altre interfacce che li hanno resi onnipresenti in questo e molti altri settori.

Per raggiungere questo risultato basterebbe anche un 8 bit con sufficienti pin e le giuste periferiche ma richiederebbe un maggiore sforzo nello sviluppo e nel mantenimento del codice. La recente diffusione di microcontrollori prestanti ne ha fatto abbassare il prezzo fino a renderli competitivi con quelli delle fasce più basse dando la possibilità di utilizzare strumenti che permettono di facilitare e velocizzare lo sviluppo del codice riducendo drasticamente il tempo necessario a introdurre nel mercato un nuovo prodotto. Si può pensare di prendere un componente che ha tutto e di impiegarlo in molte applicazioni diverse utilizzando per ogni una solo quello che serve.

Il progetto che si vuole portare a termine è abbastanza ampio e quindi il suo completo sviluppo richiede un considerevole periodo di tempo. Si seguirà quindi la prima parte di ricerca iniziale e avvio del progetto, scegliendo gli strumenti ritenuti più adatti con cui lavorare e provando le varie periferiche che si intendono utilizzare sviluppandone il software in maniera modulare. Infine si realizzerà la struttura del programma principale organizzandolo in modo da far cooperare le varie routine ed ottenere uno scheletro base al quale poter aggiungere poi i dettagli applicativi.

La trattazione che segue sarà generale e coprirà gli aspetti tecnici dello sviluppo ma senza entrare nei dettagli proprietari dovuti alla natura del lavoro. Si parlerà di microcontrollori descrivendo alcuni aspetti delle loro funzionalità e struttura generale ma la trattazione non sarà esaustiva e per una completa comprensione è richiesta la loro conoscenza.

Nei capitoli successivi si presenterà il lavoro svolto iniziando con le considerazioni fatte per la scelta del micro ed esponendo poi le sue caratteristiche di interesse partendo da quelle più semplici come le porte digitali, i timer e le loro applicazioni più comuni.

Si continuerà poi analizzando il problema del salvataggio di valori utilizzati come parametri per l'applicazione servendosi di una memorie EEPROM esterna acceduta per mezzo del bus standard I2C implementato sfruttando una periferica dedicata interna al micro.

Si procederà spiegando le potenzialità e la struttura del bus T4 evidenziando poi le problematiche riscontrate nella realizzazione del driver per la porta seriale che ne costituisce il layer fisico.

Infine verrà descritta la struttura del programma principale dalla inizializzazione dell'hardware alla configurazione delle periferiche introducendo l'uso di una macchina a stati per la gestione sequenziale delle operazioni da eseguire concludendo con alcune considerazioni sul risparmio energetico.

Capitolo 2

Scelta del micro

Il primo passo è stato quello di decidere quale microcontrollore utilizzare, e per fare questo si sono definiti dei requisiti di ricerca, cioè un insiemi di caratteristiche minime che si vogliono avere. Le componenti fondamentali di questo progetto sono il micro e le sue periferiche:

I/O digitali Per interfacciarsi con i pulsanti e i led della scheda oltre che agli ingressi o uscite digitali della scheda, come finecorsa, tasti, allarmi.

Timer Utilizzati per dare il tempo di esecuzione e sincronia al sistema (tick) e per gestire il controllo di timeout sulle seriali e sull'EEPROM.

PWM Per il controllo di luminosità in alcune uscite.

ADC Utilizzato per interpretare le informazioni che arrivano dal BlueBus, leggere la temperatura e misurare la corrente che scorre in alcune uscite.

I2C Bus seriale sincrono half duplex. Utilizzato per salvare i parametri di configurazione e lo stato attuale in una EEPROM esterna.

SCI Comunicazione seriale. Nel circuito elettrico ne sono presenti tre: per bus T4, encoder ed inverter, gestione remota.

2.1 Panoramica micro a 32bit

Nella realtà aziendale se ne utilizzano molti e in vari ambiti ma le necessità attuali richiedono di seguire le esigenze di connettività richieste dal mercato e di rendere disponibili interfacce ethernet e usb.

È di fondamentale importanza poi la rapidità di sviluppo nei progetti ottenibile riutilizzando il codice il più possibile. Questo non è immediato

quando si lavora a stretto contatto con l'hardware e raggiungibile solamente introducendo dei livelli di astrazione, affidando il controllo delle periferiche a dei driver che fanno da interfaccia. In questa visione l'utilizzo di un sistema operativo embedded real time (RTOS) rappresenta un buon approccio al problema e pertanto si intende lasciare questa strada aperta ma di non seguirla in questo progetto, introducendo le novità in maniera graduale. Ciò penalizza le prestazioni e richiede anche maggiore spazio istruzione in memoria. Si è alla ricerca quindi di un componente dalle prestazioni medio-alte capace di rispondere anche a nuove esigenze future[1], con cui lavorare ad alto livello, scalabile[2] e dal costo comparabile ai micro già in uso.

Guardando le offerte presenti nei siti web delle principali case produttrici di chip e dall'incontro con dei *field application*[3] abbiamo deciso di andare verso una soluzione a 32 bit che abbia necessariamente almeno le seguenti funzionalità:

Package: 100 pin LQFP: adeguato numero di pin mantenendo una buona facilità di montaggio

Microcontrollore a 32 bit, 80-100 MHz, 256k flash

CAN: Interfaccia (Control Area Network, Scambio di messaggi prioritari nell'automazione industriale)

USB: Interfaccia (Host, Device, On The Go)

Scalabilità verso il basso (chip più piccolo, con meno pin ma nella stessa disposizione, meno flash, meno interfaccie, più economico)

WDT e IWDT (Watch Dog Timer, classico e indipendente), Class B Compliant[4]

Stabilità dell'architettura nel tempo, assistenza, sviluppo, aggiornamenti

Consumo ridotto: Sleep mode e altre modalità di risparmio energetico

Abbiamo steso una tabella con le caratteristiche di interesse e i componenti che le posseggono, in modo da poter fare una prima comparazione (Vedi appendice A.1 pag 37). Queste sono le alternative valutate:

[1] usb, ethernet, RTOS
[2] come pinout e taglia di memoria
[3] tecnici rappresentanti di consorzi di aziende
[4] Pronto per la certificazione di sicurezza

FREESCALE [1]
DEVICE: MCF52252CAF66, coldfire
MCU: ColdFire, proprietaria
IDE: CodeWarrior
RTOS: MQX proprietario

MICROCHIP [2]
DEVICE: PIC32MX755F256L
MCU: MIPS, proprietaria
IDE: MPLAB C32
RTOS: AVIX
HITS: librerie libere (TCP/IP stacks, USB, Encrypt, file sistem, ..)

NXP Semiconductors [3]
DEVICE: LPC1768
MCU: ARM Cortex
IDE: LPCXpresso della CodeRed (basato su eclipse)
RTOS: SMX® ARM RTOS

RENESAS Electronics [4]
DEVICE: R5F562N7BNFP, RX62N
MCU: RX, proprietaria
IDE: HEW (High-performance Embedded Workshop)
RTOS: CMX, FREE RTOS, Micrium, Segger
HITS: Librerie RPDL, flash veloce quanto l'MCU, architettura CISC (RISC + DSP + FPU)

ST Microelectronics [5]
DEVICE: STM32F107VC, F107
MCU: ARM Cortex
IDE: Atollic (basato su eclipse)
RTOS: uC-OS-II RTOS from Micrium, FREE-RTOS
HITS: Librerie free, tools di terze parti a pagamento

Texsas Instruments [6]
DEVICE: LM3S9997-IQC80-C1T, stellaris
MCU: ARM Cortex
IDE: Code Composer Studio v4 (basato su eclipse)
RTOS: FREE RTOS
HITS: StellarisWare software

2.1.1 Scelta del micro

Osservando la tabella comparativa in appendice A.1, pagina 37, si può notare che le caratteristiche dei dispositivi considerati sono molto simili e per un uso generale possono essere equivalenti. Bisogna approfondire quindi le peculiarità di ciascuno per vedere se possano tornare utili nel nostro progetto.

Per prima cosa si possono distinguere due macro categorie: quelli che utilizzano un core ARM Cortex e quelli che ne usano uno proprietario. I produttori che appartengono alla prima categoria[5] vantano l'intercompatibilità del codice e la disponibilità di algoritmi utili già implementati, ma ognuno mantiene le proprie periferiche e caratteristiche particolari rendendo comunque difficoltoso un cambiamento di piattaforma, cosa comunque laboriosa. Dato poi che l'applicazione non deve svolgere particolari calcoli che richiedono l'uso di algoritmi sofisticati lo standard Cortex diventa per noi non influente.

Per quanto riguarda le pure prestazioni quello che ha il più alto rapporto di istruzioni eseguite a parità di frequenza[6] è l'RX62N della renesas, che vanta anche la memoria più veloce capace di lavorare alla stessa frequenza del core. È anche quello che possiede il maggior numero di timer disponibili e di periferiche CCP[7]. Inoltre l'ambiente di sviluppo associato consente di eseguire il debug della scheda con funzionalità avanzate e permette anche l'inserimento a caldo, cosa molto utile in fase di sviluppo e collaudo.

Anche le altre alternative sarebbero state adatte per l'applicazione ma si è deciso per quest'ultima a livello commerciale con lo scopo di aumentare l'indipendenza dai fornitori scegliendo una casa produttrice per noi nuova e leader nel settore. Fattori economici hanno spinto verso questa soluzione approfittando della campagna che sta portando avanti la renesas per promuovere questa sua nuova linea.

2.2 Presentazione del Micro

La *Renesas Electronics* è una azienda giapponese nata dalla fusione tra *NEC Electronics* e *Renesas Technology*, la quale a sua volta deriva dalla fusione tra *Hitachi* e *Mitsubishi Electric*. È una tra le aziende leader mondiali nella produzione di componenti elettronici e numero uno nei microcontrollori.

La famiglia RX600 alla quale appartiene l'RX62N utilizza una architettura harvard avanzata (fig. 2.1), con due separati bus per l'accesso a istruzioni

[5] CMSIS: Cortex Microcontroller Software Interface Standard
[6] Dhrystone MIPS, un programma di benchmark.
[7] Capture Compare PWM

e dati ma una unica memoria comune, eliminando la necessità di trasferire le costanti in RAM. Il bus istruzioni è l'unico a 64 bit per il caricamento dei comandi CISC a lunghezza variabile, utili per ridurre le dimensioni del codice. Gli altri quattro sono a 32 bit e sono dedicati al DMA (Direct Memory Access), DTC (Data Transfer Control) ed E-DMA (Ethernet Direct Memory Access) e hanno il compito di liberare il processore da onerosi e banali trasferimenti di dati, aumentando le performance complessive.

Altro punto di forza è la possibilità di eseguire moltiplicazioni e divisioni su numeri interi e l'unità Floating Point (FPU) integrata sugli stessi registri della CPU che rendono possibili impieghi in ambiti riservati ai DSP (Digital Signal Processor). È stata prestata molta attenzione alla gestione degli interrupt con la possibilità di impostare per ognuno la priorità o dedicare dei registri per velocizzare il salto e poi il ritorno dalle ISR (Interrupt Service Routine).

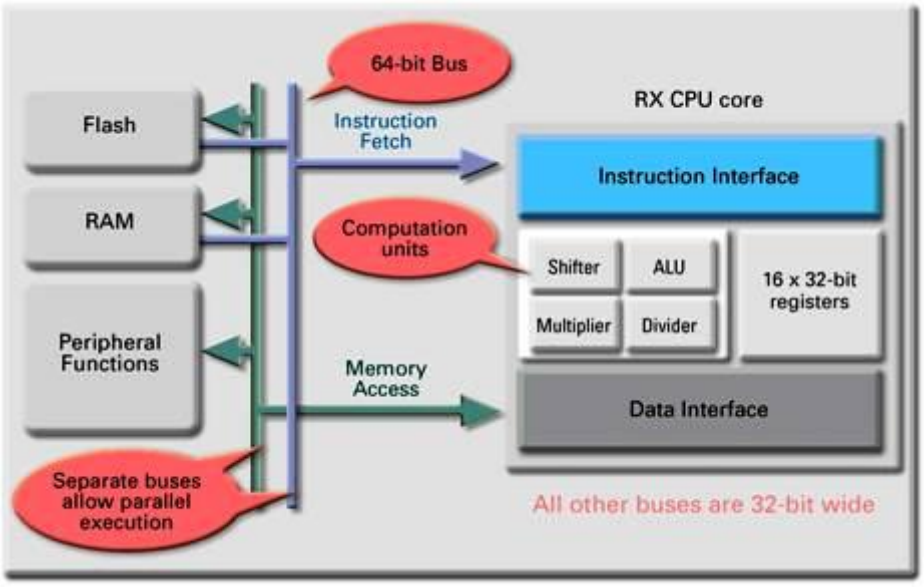


Figura 2.1: Architettura harvard avanzata

Come si può vedere dalla figura 2.2 l'insieme delle funzionalità offerte racchiude le nostre necessità. Notevole è la disponibilità e versatilità dei timer fondamentali in ogni progetto. La presenza di due differenti canali ADC permette di dedicarne uno al BlueBus[8], che richiede per una buona lettura campionature frequenti. Le periferiche di comunicazione sulle quali ci

[8]Bus dedicato alle fotocellule basato su una modulazione in corrente che richiede una campionatura analogica

soffermeremo sono l'I2C e l'SCI mentre le altre verranno valutate in progetti futuri.

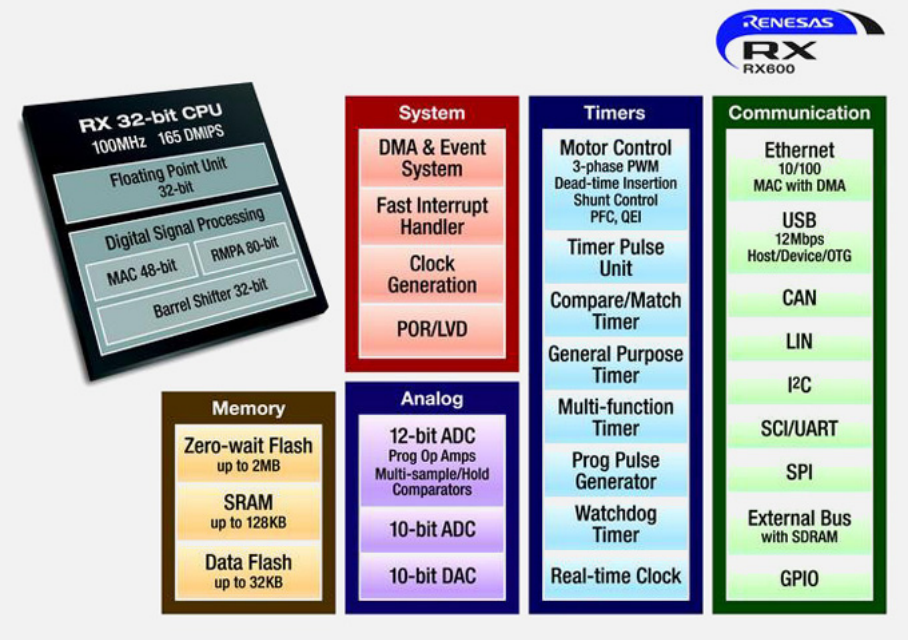


Figura 2.2: Schema a blocchi delle funzionalità

2.2.1 Le demoboard

Prima della scelta della piattaforma abbiamo provato varie demoboard e relativi ambienti di sviluppo in modo da poter valutare concretamente il materiale con cui avremmo dovuto lavorare individuando eventuali difficoltà pratiche o mancanze non considerate precedentemente.

Una demoboard è una schede elettronica dove viene montato il dispositivo di interesse nella configurazione più generale possibile e con tutti i componenti di contorno necessari al funzionamento suo e delle più comuni periferiche che vogliono essere presentate. Siccome agli stessi pin sono generalmente associate varie possibili funzionalità circuitalmente incompatibili, nella scheda sono presenti vari accorgimenti per selezionarle una alla volta tramite l'ausilio di jumper, dip-switch o componenti DNF (Do Not Fit). Prima di cominciare è quindi necessario impostarli secondo le proprie necessità.

Per verificare il funzionamento dell'I2C si è collegata una memoria EEPROM mentre per la seriale è stato necessario assemblare una piccola scheda esterna con un chip che traduce i segnali Tx e Rx in modo differenziale con lo scopo di rispettare lo standar aziendale utilizzato negli altri dispositivi con i quali si

è verificato praticamente la comunicazione. Nella fase successiva di sviluppo del protocollo Master/Slave si è resa necessaria una seconda demoboard. Di quella sulla quale è stata sviluppata l'applicazone viene riportato lo schema con la posizione dei componenti nella figura B.1 e B.1 in appendice.

2.3 L'ambiente di sviluppo

L'IDE della Renesas è fortemente orientato verso i suoi prodotti. L'acronimo HEW significa *High-performance Embedded Workshop* e oltre all'editor di testo e alle funzionalità per la compilazione e il linking comprende anche una buona gestione dei progetti in file e cartelle, il completamento automatico, la ricerca su più file, il passaggio da una funzione alla sua definizione e molti altri dettagli utili.

La Renesas offre il suo compilatore C/C++ ottimizzato appositamente per la sua architettura. Sono state riscritte anche numerose funzioni standard in modo da adattarsi alla struttura della memoria, l'uso migliore delle variabili globali e la gestione ottimale degli interrupt, tipici dei microcontrollori e non considerati nel C standard. È anche possibile scegliere la nuova versione c99 (rispetto a quella del '89), che offre al compilatore ulteriori possibilità di ottimizzazione a fronte di lievi modifiche.

Particolarità di questo IDE è quella di poterne eseguire istanze multiple e lavorare su più progetti contemporaneamente avendo così la possibilità di seguirne l'interazione.

Essendo ciò con cui si lavora l'ambiente di sviluppo è molto importante e non è costituito solamente dell'IDE ma anche dal materiale di contorno che si rivela spesso fondamentale. La documentazione del microcontrollore, gli esempi di partenza che spiegano l'utilizzo pratico delle varie periferiche, le librerie RPDL (Renesas Peripheral Driver Library) e la loro documentazione, il forum on-line dove ci si aiuta reciprocamente e si trovano nuove idee.

2.3.1 Debug

L'area di lavoro è personalizzabile ed è possibile impostare anche delle viste a seconda delle cose su cui ci si vuole focalizzare. Le vere potenzialità emergono però nella fase di prova. Una volta collegata la demoboard al computer tramite il programmatore/debugger diventa possibile caricare il firmware ed eseguirlo. In ogni momento si conosce il valore del *program counter* ed è possibile impostare dei *breakpoint* sia hardware prima dell'esecuzione che software al volo. Inoltre sono anche configurabili utilizzando dei *trigger* su eventi quali confronti o conteggi.

Ci sono finestre dedicate alla visualizzazione delle aree di memoria, di tutti i registri interni del micro suddivisi per periferica e aggiornabili su richiesta durante l'esecuzione del programma. Inoltre ogni singola variabile, elementare o composta come vettori o strutture, può essere visualizzata e mantenuta aggiornata in tempo reale. Queste possibilità sono estremamente potenti e utili nel controllo del corretto funzionamento del codice e per risolvere velocemente problemi altrimenti difficili abbreviando i tempi di sviluppo.

Ogni sorgente C può essere visualizzato in modalità assembler o mista avendo così la possibilità di mantenere sotto controllo il numero di istruzioni effettivamente tradotte e, nei casi dove ci sono alternative implementative, scegliere quella più veloce o quella che occupa meno spazio in memoria secondo necessità.

Oltre agli strumenti software si è utilizzato molto anche un oscilloscopio in quanto avevamo la necessità di controllare i livelli di tensione e le tempistiche effettivamente generate dalle periferiche, soprattutto quella seriale in particolare nella generazione del break, nell'analisi dei byte inviati e nel controllo delle collisioni.

Capitolo 3

I/O digitali e Timer

La prima cosa fatta per verificare il funzionamento del micro, del debugger e della struttura del progetto è stata quella di accendere dei led e leggere i pulsanti presenti nella demo. Si tratta di una cosa semplice ma molto importante, presente nella maggior parte delle applicazioni.

Si è subito notata la difficoltà a fare una lettura corretta degli switch e avere degli ingressi stabili a causa della loro natura meccanica. Il problema è conosciuto e la sua gestione è nota come debounce degli ingressi. Esistono vari approcci ma la maggior parte dei quali si basano sul considerare corretto un ingresso quando mantiene il suo stato per un certo periodo. È stato quindi introdotto l'utilizzo di un timer per misurare questo tempo.

Siccome il progetto prevede la possibilità di variare la luminosità di alcune luci senza l'utilizzo di una periferica dedicata [1] si è sfruttato lo stesso timer per raggiungere un risultato analogo.

3.1 Ingressi e uscite digitali

La versione a 100 pin ha 11 porte, numerate da 0 a 5 e da A a E. A ogni porta sono associati dei registri di configurazione e otto pin del miro. Appena la macchina viene accesa (o si resetta) sono tutti configurati come ingressi. In questa modalità è possibile abilitare una resistenza interna di pull-up, normalmente comune in molte applicazioni consentendo di ridurre il numero di componenti esterni necessari.

```
//Pull-Up Resistor Control Register (PCR)
PORT0.PCR.BIT.B0 = 0; //pull-up resistor off
PORT0.PCR.BIT.B0 = 1; //pull-up resistor on
```

[1] Per direttiva basata sul layout del circuito.

I pin sono condivisi con quelli delle periferiche e affinché queste possano leggere gli ingressi deve essere abilitato un buffer.

```
//input buffer control registers (ICR)
PORT0.ICR.BIT.B0 = 0; //buffer not enabled
PORT0.ICR.BIT.B0 = 1; //buffer enabled
```

Quando invece i pin sono impostati come uscita è possibile scegliere se configurarla in modo normale o *open-drain* che non assorbe corrente quando si trova nello stato alto, utile nelle comunicazioni bidirezionali.

```
//Open Drain Control Register (ODR)
PORT0.ODR.BIT.B0 = 0; //CMOS output pin
PORT0.ODR.BIT.B0 = 1; //NMOS open-drain output pin
```

Per impostare un pin come ingresso o come uscita basta specificarne la direzione tramite il registro DDR.

```
//Data Direction Register (DDR)
PORT0.DDR.BIT.B0 = 0; //input
PORT0.DDR.BIT.B0 = 1; //output
```

Per impostare il valore di un pin configurato come uscita si usa il registro DR.

```
//Data Register (DR)
PORT0.DR.BIT.B0= 0; //uscita a stato basso
PORT0.DR.BIT.B0= 1; //uscita a stato alto
```

Per leggere un ingresso invece si utilizza il registro PORT. In questo caso viene letta una intera porta e caricata in un byte.

```
char c;
c = PORT0.PORT.BYTE;
```

Nel progetto è stato dedicato un file per la gestione delle uscite digitali e uno per gli ingressi. Nella funzione di inizializzazione vengono configurati i registri secondo le necessità e in quella di aggiornamento si utilizza il solo registro per la scrittura (DR) o lettura (PORT).

Il programma principale si occupa solo di cosa deve essere scritto e il driver si occupa di farlo. Questo approccio torna utile anche qualora si dovesse adattare il progetto ad un altro micro rendendo necessario adattare il solo driver, unico posto dove si utilizzano i registri hardware. Nel caso invece si debba fare un altro progetto con la stessa piattaforma il driver può essere direttamente riutilizzato, senza badare a dettagli implementativi.

3.1.1 Debounce degli ingressi

L'incertezza degli ingressi è dovuta all'oscillazione meccanica dei pulsanti (da qui il nome debounce, antirimbalzo). La loro realizzazione sta diventando sempre più stabile e affidabile ma il problema persiste. Un modo per limitare o risolvere il problema consiste nel realizzare un filtro RC che renda più graduale il passaggio di stato, introducendo però un ritardo nella propagazione del comando.

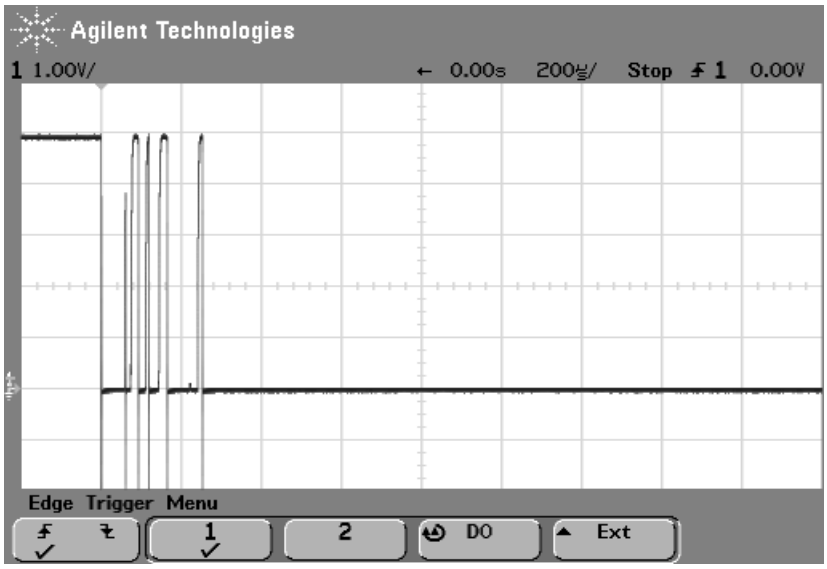


Figura 3.1: Rimbalzi di tensione alla pressione di un pulsante

Come si può vedere dall'immagine 3.1 in seguito alla pressione di un tasto la tensione associata non varia in maniera ideale a gradino ma c'è un transitorio nel quale ci sono picchi e buchi di tensione. Siccome la nostra applicazione si basa anche sulla rilevazione dei fronti dei tasti un disturbo del genere verrebbe interpretato come una sequenza di pressione e rilasci generando dei problemi.

La routine di lettura degli ingressi viene richiamata dal main a intervalli regolari e si aspetta una lettura corretta. Nel driver quindi è stata introdotta una variabile di appoggio che contiene il valore dell'ultima lettura e un contatore che viene incrementato se non ci sono state variazioni. Quando il contatore supera un certo valore configurabile allora l'ingresso è considerato stabile e viene passato all'applicazione, che è trasparente a tutto questo.

3.2 Timer

Questo micro ha una vasta scelta di timer di vario tipo che lo rendono estremamente versatile e permette di dedicarne uno per ogni esigenza, senza andare al risparmio.

Si dividono in tre categorie, accessibili come periferiche distinte (vedi figura 3.2). Ogni una è divisa in due unità indipendenti che possono essere abilitate quando necessario o rimanere spente per risparmiare energia. Tutte derivano il proprio *clock* da quello comune alle periferiche tramite dei *divisori di frequenza* configurabili da dei registri. Vedi figura A.2 in appendice.

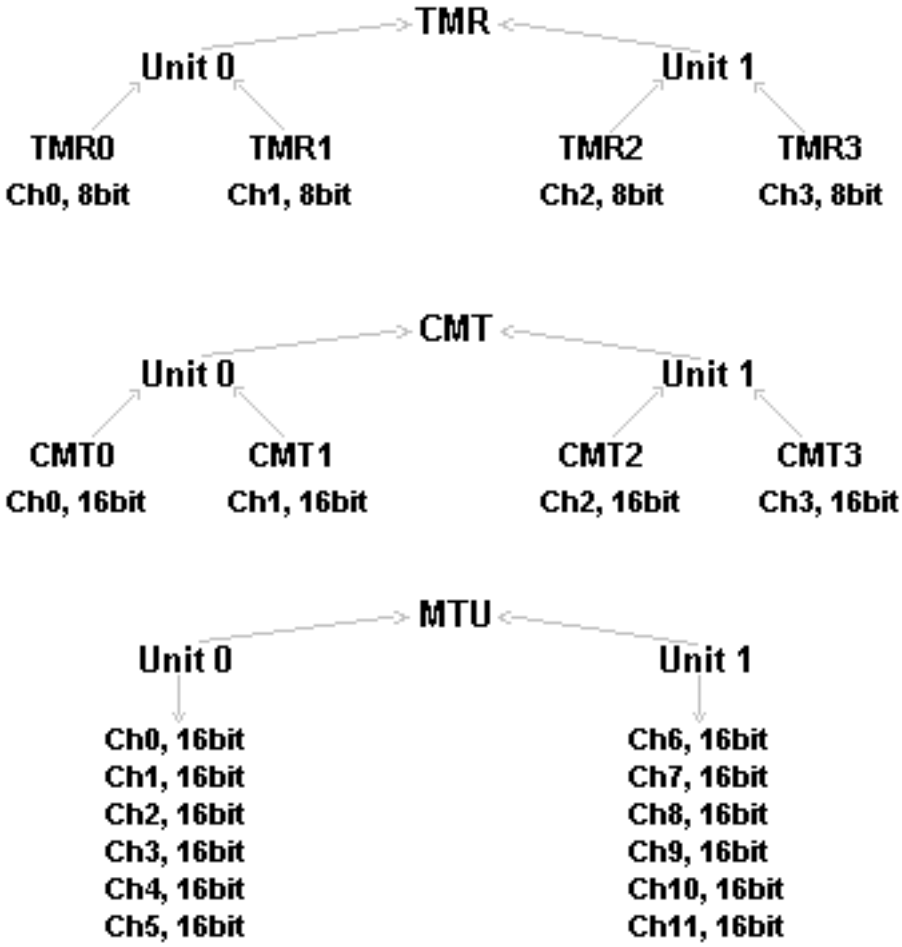


Figura 3.2: Struttura dei timer

TMR 8-bit timers, (8 bits x 2 channels) x 2 units

Si possono mettere in cascata due timer per ottenerne uno da 16 bit.

Interrupt sul confronto e overflow.
Clock selezionabile anche da sorgente esterna.

CMT Compare match timer, (16 bits x 2 channels) x 2 units
Interrupt sul confronto.

MTU Multi-function timer pulse unit, (16 bits x 6 channels) x 2 units
Azzeramento dei contatori contemporaneo
Lettura della frequenza di ingresso
Controllo di motori brushless in PWM complementario
Uscita della forma d'onda al confronto

Ad esempio per configurare un timer a 8 bit per dare un interrupt ogni millisecondo si può seguire il codice sottostante.

Dopo aver abilitato l'unità comprendente il timer che si intende utilizzare si disabilitano gli interrupt e si azzera il contatore. Poi, sapendo che il clock delle periferiche è a 48MHz, si seleziona il divisore adatto a ottenere una frequenza di 1KHz moltiplicata per un valore a 8 bit. Gli unici divisori che permettono di avere $0 < TCORA < 256$ secondo l'appendice A.2 sono 1024 e 8192 ma abbiamo scelto il primo per avere una risoluzione maggiore e per non caricare troppo i *flip-flop* del divisore in modo che consumino meno. Vedi la formula **??**.

Infine si abilita l'interrupt quando il valore in TCORA corrisponde a quello del contatore che verrà contestualmente azzerato.

$$\begin{aligned} \frac{PCLK}{DIV} &= \frac{1}{1ms} \cdot TCORA \\ \frac{48MHz}{1024} &= 1KHz \cdot TCORA \\ TCORA &= \frac{48MHz}{1024 \cdot 1KHz} = 47 \end{aligned} \tag{3.1}$$

```
void Timer0_Init(void)
{
  MSTP(TMR0)=0; //Attiva periferica Timer0
  IEN(TMR0,CMIA0)=0; //interrupt compare match A disabilitato
  IEN(TMR0,CMIB0)=0; //interrupt compare match B disabilitato
  IEN(TMR0,OVI0)=0; //interrupt overflow disabilitato
  TMR0.TCNT=0; //Resetta il contatore

  TMR0.TCORA=47; //valore per il match A, 48MHz/1024/47=1ms
  TMR0.TCCR.BYTE=0x0D; //divisore PCLK/1024
```

```
  TMR0.TCR.BYTE=0x48; //clear on match A, interrupt on match A

  IPR(TMR0,CMIA0)=0x04; //priorità
  IR(TMR0,CMIA0)=0; //clear interrupt flag
  IR(TMR0,CMIB0)=0; //clear interrupt flag
  IR(TMR0,OVI0)=0; //clear interrupt flag
  IEN(TMR0,CMIA0)=1; //interrupt compare match A abilitato
}
```

3.2.1 Base tempi del sistema

Molte operazioni necessitano di essere eseguite a intervalli regolari oppure serve gestire dei tempi relativamente lunghi durante l'interazione con l'utente. Serve quindi una infrastruttura per temporizzare in modo indipendente, elegante ed efficiente tutte le funzioni che lo necessitano. Viene chiamato *tick timer*.

Il vantaggio è quello di utilizzare un unico timer hardware per servire molte risorse. Viene realizzato incrementando in interrupt un contatore e rendendo disponibili due funzioni: una che ritorna il valore di quel contatore e una che dà la differenza tra quel contatore e un'altro passato come parametro e gestito dall'utente di queste routine.

Per rendere le cose più eleganti e facilmente portabili anche su altre piattaforme[2] si è deciso di definire un nuovo tipo di dato e di usarlo sempre per specificare il contatore. Si tratta di una variabile intera a 32 bit[3] senza segno. Questo dettaglio è importante perché consente di avere sempre un valore corretto sia in caso di *overflow* che nelle *differenze*, come evidenziato in questo esempio riportato sugli 8 bit.

$$255 + 1 = 0$$

$$25 - 20 = 5$$

$$20 - 250 = 15$$

```
typedef unsigned long int TICK;
static TICK tick = 0;

void tick_timer()  //da richiamare in interrupt
{
  tick++;
```

[2] Ad esempio non a 32 bit.

[3] In questa architettura

```
}

TICK tick_current()
{
  return tick;
}

TICK tick_elapsed(TICK t)
{
  return (tick - t);
}
```

Specificando una firma del genere si inserisce la funzione *tick_timer* nell'*interrupt vector* alla posizione assegnata al *match A* del *timer 0*, come inizializzato nella sezione precedente.

```
#pragma interrupt tick_timer(vect=VECT_TMR0_CMIA0)
void tick_timer();
```

L'utilizzo poi è intuitivo e molto versatile. Compilato con le ottimizzazioni dà del codice molto contenuto, si utilizza però una variabile per ogni funzione ma questo non è un problema considerato la taglia di memoria del microcontrollore.

```
TICK t1, t2;
while(0) //main
{
 //..
 if(tick_elapsed(t1)>10)
 {
  t1=tick_current();
  //fa qualcosa ogni 10ms
 }
 //..
 if(tick_elapsed(t2)>1000)
 {
  t2=tick_current();
  //fa qualcosa ogni secondo
 }
 //..
}
```

3.3 PWM

In questa applicazione viene richiesto di poter variare la luminosità di una luce [4] agendo solamente sulla sua abilitazione. Se una lampada viene accesa e spenta abbastanza velocemente l'effetto risultante è quello di una sua attenuazione proporzionale alla percentuale di tempo in cui rimana accesa piuttosto che spenta. Questa percentuale viene detta *duty cycle* mentre il numero di cicli effettuati al secondo e detta frequenza anche se si usa spesso il suo reciproco, detto periodo. Vedi figura 3.3. Questa tecnica è conosciuta come *Pulse Widh Modulation* e consiste appunto nel generare degli impulsi di lunghezza variabile. È ampiamente utilizzata per variare la potenza trasferita a un carico.

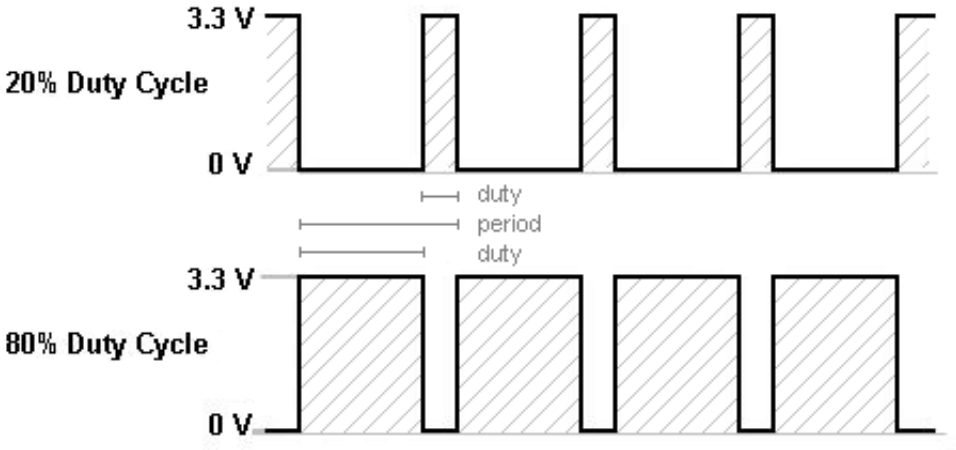


Figura 3.3: Variazione della potenza trasmessa tramite PWM

Per specifica non è stato possibile utilizzare un modulo dedicato e si è dovuto implementare questa modulazione utilizzando un semplice timer a 8 bit.

Nella funzione di inizializzazione si imposta il timer sfruttando entrambi i registri per il *compare match*. All'uguaglianza del primo viene generato un interrupt che determina la fine del periodo attivo, l'uguaglianza del secondo determina sia l'azzeramento del contatore che il termine del periodo di inattività. La frequenza dipende sia dal valore del clock delle periferiche e da quello del divisore che dal secondo registro di confronto del timer.

Il carico per il processore è minimo in quanto le ISR vengono richiamata solo quando è necessario cambiare lo stato dell'uscita e sono molto leggere.

[4] lampada a incandescenza o a led abilitata tramite un MOSFET.

La frequenza di modulazione scelta è bassa rispetto a quella del processore e delle periferiche.

```
#define TMR1_SCALER 0x0E //PWM_FREQ=PCLK/8192/TMR1_PERIOD
#define TMR1_PERIOD 100

void Out_Init(void)
{
  //..
  TMR1.TCORA=0; //match con A, rappresenta il duty
  TMR1.TCORB=TMR1_PERIOD; //match con B, 100 divisioni nel periodo
  TMR1.TCR.BYTE=0xD0; //clear on match B, interrupt on match A e match B
  //..
}

void Out1_Set_Duty(BYTE duty)
{
  TMR1.TCORA = duty*TMR1_PERIOD/100; //proporzione
}

void Out1_On(void)
{
  TMR1.TCCR.BYTE = TMR1_SCALER;
}

void Out1_Off(void)
{
  TMR1.TCCR.BYTE = 0;
  PORTD.DR.BIT.B5 = 0;
}

//ISRs
#pragma interrupt Interrupt_TMR1_CMA(vect=VECT_TMR1_CMIA1)
void Interrupt_TMR1_CMA(void)
{
  if(TMR1.TCORA!=TMR1.TCORB) PORTD.DR.BIT.B5 = 0;
}

#pragma interrupt Interrupt_TMR1_CMB(vect=VECT_TMR1_CMIB1)
void Interrupt_TMR1_CMB(void)
{
```

```
  PORTD.DR.BIT.B5 = 1;
}
```

Siccome il progetto prevede due uscite di questo tipo può capitare che esse vengano accese contemporaneamente e questo determina una corrente di spunto notevole che l'alimentatore deve erogare. Per limitare questo problema si può sfasare una delle uscite in modo tale che i fronti di salita non coincidano, oppure utilizzare due frequenze diverse.

Capitolo 4

EEPROM su bus i2c

La centrale deve offrire la possibilità di essere configurata personalizzando tutta una vasta serie di opzioni per adattala alle molteplici esigenze che il mercato richiede. I parametri comprendono le informazioni riguardanti la rete, i dispositivi associati, il comportamento da seguire quando in manovra e lo storico degli eventi.

Alcune di queste informazioni (come il MAC address) vengono generate ed inserite nella fase produttiva e per questo risulta comodo utilizzare una memoria esterna, resa facilmente accessibile e sostituibile per poter ovviare anche a eventuali problemi di usura dovuti al limitato numero di scritture intrinseco alla tecnologia.

Componenti con esattamente queste proprietà sono le EEPROM, piccoli chip di memoria accessibili tramite I2C, un bus seriale che necessita di due soli fili e utilizzabile anche per altri dispositivi analoghi.

Si è strutturato questa parte realizzando vari strati in modo da ottenere l'indipendenza dall'hardware. La gestione del bus i2c ha a che fare con i livelli di tensione e le tempistiche e può essere implementato completamente in software o sfruttando una periferica dedicata. La lettura/scrittura sulla EEPROM viene eseguita mandando opportune serie di comandi sul bus resi trasparenti da delle funzioni che lavorano su vettori. Il salvataggio dei parametri si preoccupa solo della struttura che i dati devono avere, rendendo disponibili delle funzioni per il loro reperimento o memorizzazione.

Si è ottenuto così sia *l'indipendenza fisica* dei dati, che possono essere memorizzati in qualsiasi supporto e seguendo una qualsiasi struttura, che *logia* in quanto l'utilizzatore di queste funzioni accede allo stesso modo ai dati anche se questi possono essere organizzati in maniera diversa. Questa proprietà è tipica dei sistemi di gestori di basi di dati[1].

[1] DBMS

4.1 Il bus i2c

Il bus I2C (Inter-Integrated Circuit) è uno standard introdotto dalla Philips[7] all'inizio degli anni '90 per far comunicare vari circuiti integrati montati sulla stessa scheda tra di loro. Lo scopo è quello di limitare il numero di piste necessarie semplificando quindi la progettazione e riducendo le dimensioni. Sono necessarie solamente due linee bidirezionali open-drain[2]: una per i dati (SDA) e una per il clock (SCL), rendendo il protocollo sincrono e quindi ancora più semplice. Il bus è popolato da due possibili categorie di dispositivi: *master* e *slave*, ma in questa e nella maggior parte delle implementazioni esiste un solo master, ed è l'unico a poter iniziare una comunicazione per eseguire delle interrogazioni. Ogni slave ha un indirizzo a cui risponde secondo le proprie caratteristiche solo in seguito a una richiesta. In queste condizioni le collisioni vengono evitate.

Questo driver è stato implementato sfruttando la periferica hardware dedicata del microcontrollore, ma esistono anche alternative, dette *bit-banging*, che si basano sul controllo diretto delle tempistiche e delle linee. L'utilizzo è semplice attraverso le funzioni in stile *read/write* che lavorano su sequenze di byte individuati da un puntatore e dalla loro lunghezza. Sono stati introdotte delle particolarità per facilitare e rendere più efficiente l'accesso da parte del modulo di gestione dell'EEPROM, come la funzione *verify* e la revisione di write utilizzando due buffer.

Tutte le funzioni sono bloccanti, cioè non terminano finché non hanno completato il loro compito. Questo è necessario perché l'applicazione non può continuare se prima l'accesso alla memoria è avvenuto correttamente. In caso di errori bisogna ritornare in una condizione consistente con altre procedure. Nelle parti nelle quali si deve rimanere fermo in *busy waiting* c'è comunque un sistema di timeout che gestisce l'eccezione.

Il valore di ritorno specifica se ci sono stati degli errori e permette la loro gestione. I parametri comprendono l'indirizzo del slave che si vuole interrogare, il puntatore al buffer dove leggere o scrivere i dati e il loro numero. La procedura di inizializzazione abilita la periferica, seleziona i pin di uscita e le impostazioni per il clock. "Device_Ready" verifica se un dispositivo risponde.

```
#define I2C_NO_ERR 0 //nessun errore
#define I2C_ERR_TIMEOUT 4 //timeout
#define I2C_ERR_BUS 8 //errore sul bus
#define I2C_ERR_COMM 16 //errore di comunicazione
#define I2C_ERR_BUSY 32 //device busy
#define I2C_ERR_VERIFY 64 //verify failed
```

[2]I segnali sono normalmente alti e vengono abbassati da uno dei componenti sul bus.

```
void RIIC0_Init (void);

char RIIC0_Master_Read(unsigned char addr_dev,
    unsigned char  *p_data,
    unsigned long len,
    unsigned char start);

char RIIC0_Master_Verify(unsigned char addr_dev,
    unsigned char  *p_data,
    unsigned long len,
    unsigned char start);

char RIIC0_Master_Write(unsigned char addr_dev,
    unsigned char *p_data,
    unsigned long len,
    unsigned char stop);

char RIIC0_Master_DB_Write(unsigned char addr_dev,
    unsigned char *p_addr_buf,
    unsigned long len_addr_buf,
    unsigned char *p_data_buf,
    unsigned long len_data_buf,
    unsigned char stop);

char RIIC0_Device_Ready(unsigned char addr_dev);
```

In figura 4.1 viene proposto un esempio di scrittura secondo il protocollo i2c. Dopo il bit di START il master invia il *control byte* formato dall'indirizzo del dispositivo (7 bit) e da un bit che specifica il tipo di operazione voluta: 0=Scrittura, 1=Lettura. Al termine dell'invio il controllo della linea passa allo slave che riconosce l'indirizzo e mantenendola bassa per il tempo di un bit dichiara la corretta ricezione e abilita il master a continuare la trasmissione (acknowledge, ACK).

Il secondo byte rappresenta l'indirizzo della locazione di memoria sulla quale si desidera scrivere che verrà copiato nell'*address pointer* del dispositivo. Il byte successivo è il dato trasmesso che viene salvato nella cella di memoria puntata dall'address pointer che viene contestualmente incrementato. Lo slave manda l'ACK e se il master genera lo STOP la trasmissione finisce, altrimenti rimane in attesa di un nuovo dato da memorizzare (sfruttando l'incremento automatico dell'address pointer).

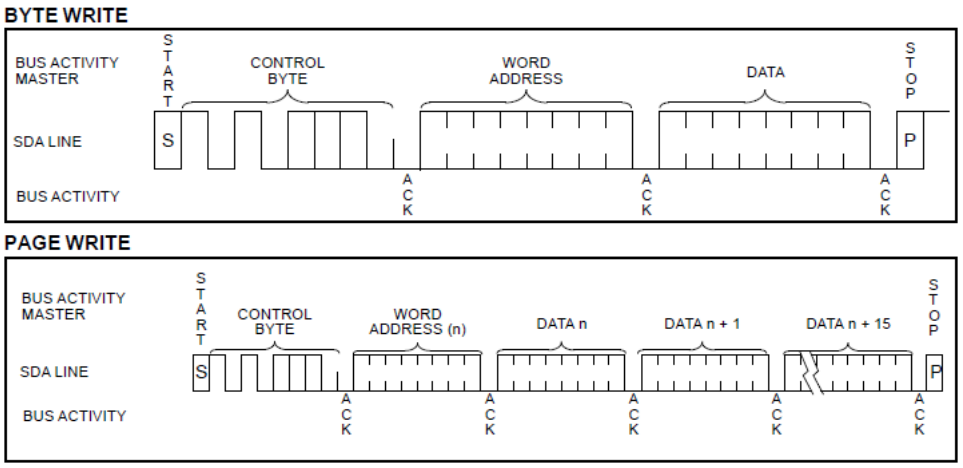


Figura 4.1: Sequenza di byte trasmessi in un tipico accesso di scrittura.

Tutto questo viene gestito della funzione "RIIC0_Master_DB_Write" che usa due buffer proprio per separare il word address (che può essere maggiore di un byte) dalla parte dati.

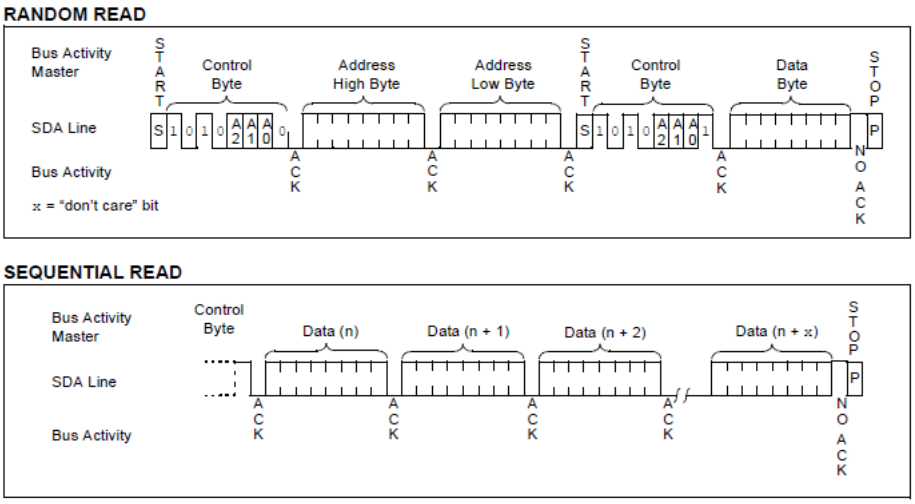


Figura 4.2: Sequenza di byte trasmessi in un tipico accesso di lettura.

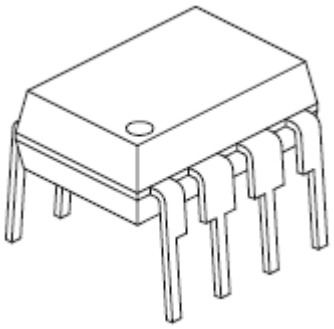
In figura 4.2 si vede come avviene la lettura, si nota che è più complessa e divisa in due fasi: una prima di scrittura per memorizzare nell'address pointer l'indirizzo della cella da leggere, e la seconda dove avviene effettivamente la lettura.

Nella seconda fase il control byte ha l'LSB messo a uno e questo specifica che l'operazione richiesta è di lettura, il dispositivo quindi invia il primo

dato e incrementa il puntatore. Se il master lo riconoscere (ACK) allora procede nell'invio del byte successivo altrimenti (NACK) termina e si aspetta la generazione dello STOP.

Questo si ottiene usando prima la funzione "RIIC0_Master_DB_Write" specificando l'opzione di non generare il bit di STOP e poi "RIIC0_Master_Read" con l'opzione di generare un RESTART.

4.2 EEPROM



Le memorie EEPROM (Electrically Erasable Programmable Read-Only Memory) seriali sono individuate da un indirizzo a 7 bit fisso, parte del quale è spesso riportato su alcuni pin del chip e quindi impostabile a livello di progettazione[3]. A volte lo stesso componente risponde a vari indirizzi rendendo accessibile uno spazio di memoria normalmente non indirizzabile utilizzando un address word composto da un unico byte. Questa suddivisione viene detta a *blocchi*.

Internamente sono divise in *pagine* la cui dimensione costante rappresenta il numero massimo di accessi sequenziali che si possono fare. Questo è legato al metodo di incremento dell'address pointer e alla dimensione del buffer interno.

Nel modulo di gestione si sono definite delle costanti che permettono di specificare queste caratteristiche in modo da poter utilizzare memorie con capacità e struttura diverse.

```
#define EE_PAGE_SIZE 128
#define EE_ADDRESS_BYTE_NUMBER 2
#define EE_BLOCK_NUMBER 2
#define EE_PAGE_PER_BLOCK 512
```

[3]In modo da far coesistere chip uguali sullo stesso bus

```
#define EE_SEQ_READ_BOUND 65536

device_address[0] = 0xA0;
```

L'interfaccia richiesta deve mascherare completamente la struttura interna e fornire metodi di accesso per byte e per vettore. Questo significa che la scrittura di un buffer a partire da un certo indirizzo venga comunemente tradotta in *molti accessi* a pagine consecutive e richiede la gestione dei tempi di svuotamento del buffer dei dispositivi durante il quale *non rispondono alle richieste*.

Si sono definiti dei tipi di dato per rendere più comprensibili e portabili le funzioni. Il valore di ritorno specifica se ci sono stati degli errori e offre la possibilità di gestirli. "EEPROM_Verify_Array" e "EEPROM_Write_Array_Verify" sfruttano "RIIC0_Master_Verify" per controllare che i dati siano stati scritti in modo corretto. È stata creata questa funzione per limitare l'utilizzo di buffer e chiamate inutili.

"EEPROM_Write_Array_FF" ha il compito di svuotare completamente la memoria assicurandosi che sia riportata alle condizioni di fabbrica.

"EEPROM_Ready" richiama direttamente la rispettiva "RIIC0_Device_Ready", per vedere se un dispositivo risponde e capire se è presente, assente, disponibile o occupato in un ciclo di scrittura del buffer interno.

```
typedef unsigned long int EE_ADDR;
typedef unsigned long int EE_LENGTH;
typedef unsigned char EE_DATA;

#define EE_BUF_SIZE 256
#define EE_TIMEOUT  10 //timeout in milliseconds
#define EE_NO_ERR 0 //tutto ok
#define EE_ERR_ADDR 1 //indirizzo non valido
#define EE_ERR_LENGTH 2 //numero di dati da scrivere non valido
#define EE_ERR_TIMEOUT 4 //Timeout error
#define EE_ERR_BUS  8 //errore sul bus
#define EE_ERR_COMM  16 //errore di comunicazione
#define EE_ERR_BUSY  32 //device busy
#define EE_ERR_VERIFY 64 //verify failed

void EEPROM_Init();
char EEPROM_Read_Byte(EE_ADDR addr, EE_DATA *dato);
char EEPROM_Write_Byte(EE_ADDR addr, EE_DATA dato);
char EEPROM_Write_Byte_Verify(EE_ADDR addr, EE_DATA dato);
```

```
char EEPROM_Read_Array(EE_ADDR addr, EE_DATA buf[], EE_LENGTH length);
char EEPROM_Write_Array(EE_ADDR addr, EE_DATA buf[], EE_LENGTH length);
char EEPROM_Verify_Array(EE_ADDR addr, EE_DATA buf[], EE_LENGTH length);
char EEPROM_Write_Array_Verify(EE_ADDR addr, EE_DATA buf[],
        EE_LENGTH length);
char EEPROM_Write_Array_FF(EE_ADDR addr, EE_DATA buf[], EE_ADDR length);
char EEPROM_Ready();
```

La gestione della suddivisione in blocchi può essere sfruttata per far apparire contiguo lo spazio di memoria ottenuto dall'unione di molti dispositivi diversi purché siano simili, cioè con la stessa struttura interna.

4.3 Salvataggio parametri

L'applicazione ha la necessità di salvare molti parametri appartenenti ad aree concettualmente diverse. Per organizzare in maniera opportuna i dati si è pensato di ricorrere al costrutto della struttura del C. Si è quindi definito un nuovo tipo di struttura (PARAM) composto internamente in molte altre strutture, una per area, contenenti campi che rappresentano le variabili da salvare.

La funzioni di inizializzazione resetta i valori della *variabile statica* "param" di tipo "PARAM"[4]. L'accesso con la notazione puntata rende intuitivo reperire il valore cercato e mantiene visibilmente separate le varie aree.

L'accesso tramite buffer offerto dall'interfaccia per l'EEPROM rende possibile trattare non solo vettori ma anche qualsiasi area di memoria. Basta infatti passare l'indirizzo del primo byte e il loro numero totale.

La funzione "Param_Load_Default" carica nella variabile statica i valori iniziali standard che assicurano il corretto funzionamento del programma. Viene richiamata in caso di problemi di lettura.

La funzione "Param" ritorna il puntatore alla struttura di memoria attraverso il quale è possibile accedere ai parametri.

```
typedef struct{
//..
}PARAM;

void Param_Init(void);
void Param_Load_Default(void);
void Param_Read_EE(void);
```

[4]Il compilatore è case sensitive

```
void Param_Write_EE(void);
PARAM* Param(void);
```

Le funzioni di lettura e scrittura richiamano direttamente[5] quelle di accesso all'EEPROM, specificando un indirizzo di prova. Questa parte sarà completata introducendo dei controlli di integrità della struttura ed evitando di riscriverla sempre nello stesso punto ma traslandola continuamente in modo da sfruttare la vita di tutte le celle.

```
EEPROM_Read_Array(64, &param, sizeof(param));
EEPROM_Write_Array(64, &param, sizeof(param));
```

A seconda del tipo dei campi delle strutture viene riservato in memoria un congruo numero di byte, ma siccome l'architettura è a 32 bit il compilatore non li dispone sempre in modo contiguo ma cerca di non far accavallare una variabile tra due *word*, in modo da recuperarla con un unico accesso, come in figura 4.3. Questo fa parte delle ottimizzazioni e normalmente è utile ma in

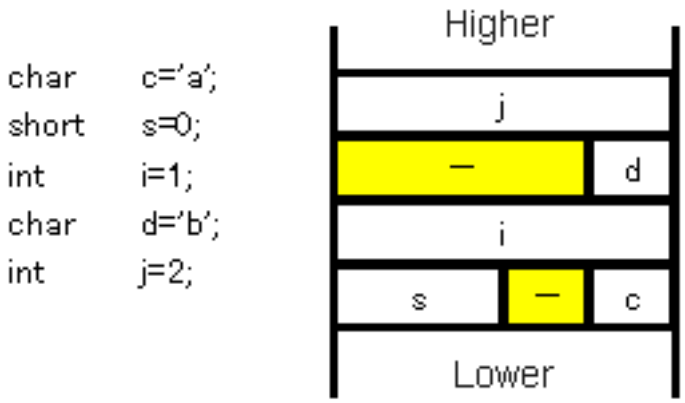


Figura 4.3: Presenza di byte di padding.

questo caso particolare va evitato. Siccome non è pratico e sempre fattibile mantenere la struttura ordinata come in figura 4.4 si è dovuto disabilitare questa funzionalità sfruttando le direttive *pragma* del compilatore.

```
#pragma pack
  //dichiarazione della struttura
#pragma unpack
```

[5]Con i dovuti controlli di errore

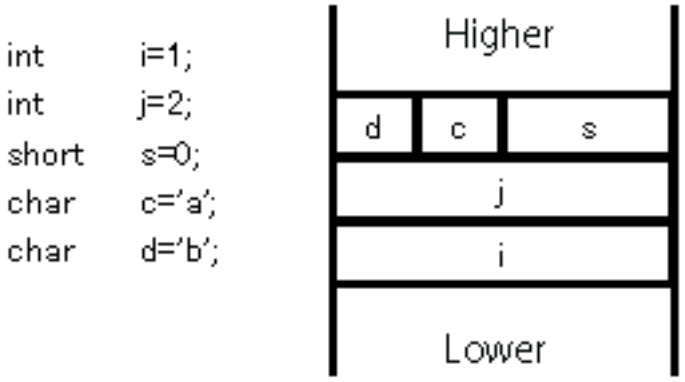


Figura 4.4: Eliminazione dei byte di padding.

Capitolo 5

Serial Communications Interface

Ogni microcontrollore che si rispetti deve permettere la gestione di una comunicazione seriale. È senza dubbio l'interfaccia più diffusa nelle sue varie implementazioni e rappresenta la norma per la maggior parte delle schede, bus, computer, alcune memorie[1] ....

L'RX62N dispone di sei unità indipendenti di questo tipo con una vasta gamma si opzioni e funzionalità. Sono configurabili in varie modalità: *sincrona*, *asincrona* e *smart card*. In generale a una connessione sono associate tre linee elettriche (vedi figura 5.1) che servono per la ricezione, la trasmissione e la sincronia. Alle linee dati (Rx e Tx) sono associati due *Shift Register* (RSR e TSR) attraverso i quali viene *spostato* nel/dal pin relativo il valore attuale sincronizzato tramite la linea di clock (SCK). Una volta che una trama completa è stata trasferirà il byte viene copiato nel/dal *Data Register* e generato un interrupt (RXI o TEI) per la sua gestione. Attraverso questo meccanismo di *doppio buffer* è possibile ricevere e trasmettere in modo continuo e contemporaneo.

La linea di clock non è obbligatoria, se presente la comunicazione viene detta sincrona altrimenti asincrona. In questo ultimo caso tutti i vari partecipanti devono avere un oscillatore interno sufficientemente preciso e stabile per garantire una corretta interpretazione dei segnali. Ogni unità ha per questo un blocco destinato alla generazione del baud rate derivato dalla frequenza comune alle periferiche.

Affidare l'informazione al livello di tensione di una linea elettrica risulta in alcune condizioni poco affidabile. Per questo quando si opera in ambi-

[1] Ad esempio SD, Secure Digital.

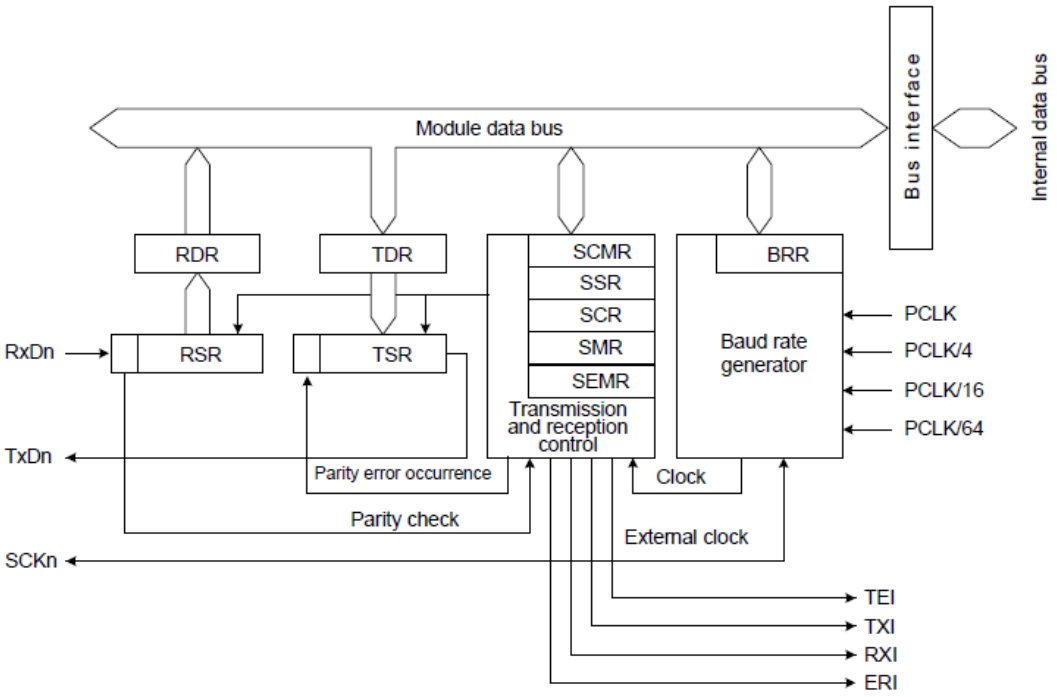


Figura 5.1: Schema a blocchi della periferica.

enti particolarmente rumorosi[2], come quelli industriali, diventa necessario legare l'informazione alla *differenza* di tensione, grandezza più difficilmente alterabile da un potenziale disturbo che agirebbe su entrambe le linee.

Siccome l'applicazione è stata testata su un bus di questo tipo è stato necessario realizzare un adattatore di livello montando un piccolo circuito elettronico, che sfrutta un componente comunemente utilizzato nelle interfacce CAN, che ha proprio questo compito. Il chip ha soli otto pin e richiede pochi componenti di contorno come in figura 5.2. Si può utilizzare il `MAX3058` della maxim o l'equivalente `MCP2551` della microchip.

Il transceiver introduce anche la funzionalità di *loop back*, che riproduce in ricezione tutto quello che viene mandato in trasmissione. Questo viene sfruttato per verificare la correttezza delle trame inviate e riconoscere alcuni problemi hardware.

[2]Dal punto di vista elettrico, ad esempio in presenza di motori

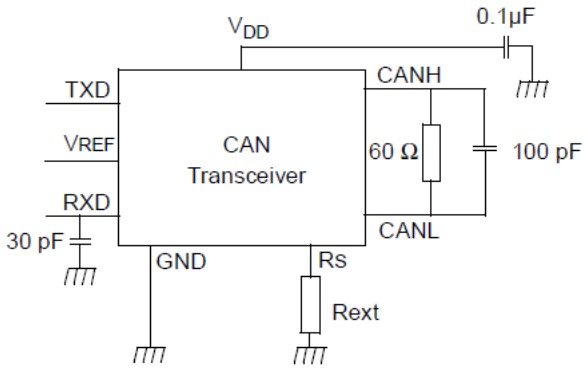


Figura 5.2: Schema di un transceiver CAN.

5.1 Seriale asincrona

Nella modalità asincrona vengono usate le sole linee di ricezione e trasmissione. In entrambi i casi la trama dei dati segue la struttura generale della figura 5.3, dove si nota che il livello normale di inattività corrisponde allo stato alto (idle in mark state).

Quando la linea viene abbassata (space state, low level) si riconosce lo *start bit* che avvisa l'inizio di una comunicazione. Seguono i bit dei dati, che possono essere sette o otto, poi uno eventuale di parità (pari o dispari) e uno o due di stop. In seguito la linea torna libera e può cominciare subito una nuova trama con un altro start bit.

Il modulo ha una unità di controllo di errore che genera un interrupt nel caso il bit di parità non corrisponda oppure che i dati non rispecchino la struttura della trama.

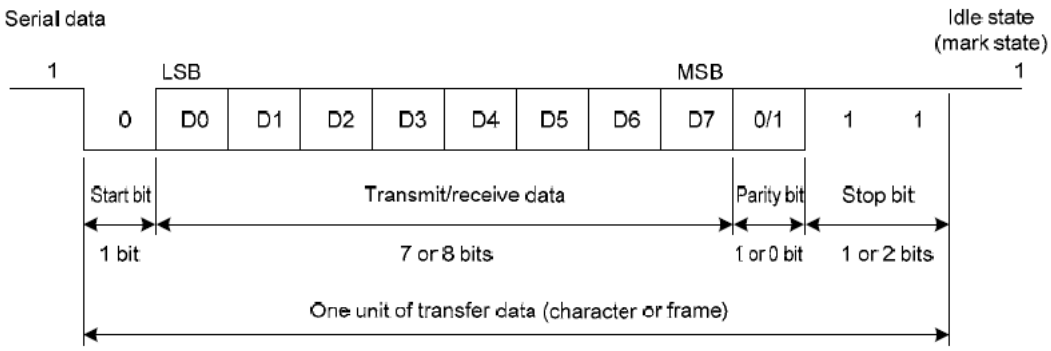


Figura 5.3: Trama asincrona: 8 bit dati, parità, 2 bit stop.

Alcuni protocolli prevedono l'invio di pacchetti di dati preceduti da un

break, cioè viene mantenuta la linea bassa per la durata di 11–13 bit con lo scopo di segnalare l'arrivo di un nuovo dato e dare il tempo ai ricevitori di avviare e sincronizzare l'oscillatore o interrompere una trasmissione in corso. Questo viene rilevato andando a leggere direttamente il valore del pin in caso di un errore sulla trama.

In seguito alla ricezione di un byte questo viene trasferito dal *Receive Shift Register* al *Receive Data Register* e generato l'interrupt *Receive data full* affinché venga tempestivamente letto. Se questo non avviene prima dell'arrivo del byte successivo si ha una sovrapposizione e la conseguente perdita di un valore, segnalata da un interrupt di errore di *overrun*.

I registri necessari alla configurazione sono i seguenti:

SMR Serial mode register
Parità, numero di stop bit e divisore per PCLK (n in **??**).

BRR Bit rate register
Determina la frequenza della trasmissione, secondo la formula **??**.

SCR Serial control register
Abilita selettivamente ricezione, trasmissione e gli interrupt.

TDR Transmit data register
Immagazzina il prossimo dato da trasmettere.

SSR Serial status register
Flag degli interrupt ed errori.

RDR Receive data register
Immagazzina l'ultimo byte ricevuto.

SCMR Smart card mode register

SEMR Serial extended mode register

$$BRR = \frac{PCLKFrequency}{64 \cdot 2^{2n-1} \cdot BitRate} - 1 \tag{5.1}$$

$$BRR = \frac{48 \cdot 10^6}{64 \cdot 2^{2 \cdot 0-1} \cdot 19200} - 1 = 77$$

5.2 Struttura e gestione a interrupt

Alle comunicazioni seriali sono associati quattro sorgenti di interrupt, abilitabili tramite il registro SCR, e i relativi flag in SSR. Sfruttando questa caratteristica è possibile implementare le ISR[3] in modo che la maggior parte del carico di gestione sia richiamato solo quando necessario e ridotto al minimo. Gli interrupt sono i seguenti:

RXI *Receive data full.* Se in ricezione segnala un nuovo dato in RDR.

TXI *Transmit data empty.* Se in trasmissione segnala lo spostamento del dato da TDR a TSR.

ERI *Receive Error.* Segnala un errore, discriminabile tramite i flag.

TEI *Transmit End.* Registro TDR non aggiornato, fine trasmissione.

Seguendo la logica della trasmissione e sfruttando le potenzialità offerte si sono implementare delle routine che realizzano la comunicazione seguendo le regole del protocollo interno aziendale e gestendo eventuali situazioni di errore. Il codice prodotto è risultato piuttosto elaborato ma l'idea di base è riassumibile facilmente dal codice seguente e osservando la figura 5.4.

Per prima cosa si configura la periferica impostando l'oscillatore e la struttura della trama. Quando si vuole spedire una serie di byte si abilita la trasmissione che genera subito un TXI, segnalando che il buffer è vuoto. Questo richiama la ISR relativa a TXI nella quale viene copiato nel registro TDR il valore da spedire e, una volta trasmesso, la routine viene nuovamente richiamata permettendo al byte successivo di essere caricato. Se l'aggiornamento non avviene in tempo significa che non ci sono nuovi valori e viene generato un TEI nella cui ISR la trasmissione viene disabilitata.

La ricezione può rimanere sempre abilitata, dopo che un nuovo byte è stato ricevuto correttamente viene generato un RXI che permette di reperirlo subito e processare.

```
#pragma interrupt Interrupt_SCI6_RXI6(vect=VECT_SCI6_RXI6)
void Interrupt_SCI6_RXI6(void)
{
  unsigned char dato;
  dato = SCI6.RDR;
}
```

[3]Interrupt Service Routine

```
#pragma interrupt Interrupt_SCI6_TXI6(vect=VECT_SCI6_TXI6)
void Interrupt_SCI6_TXI6(void)
{
  SCI6.TDR = buffer[state++];
}

#pragma interrupt Interrupt_SCI6_ERI6(vect=VECT_SCI6_ERI6)
void Interrupt_SCI6_ERI6(void){} //gestione errori

#pragma interrupt Interrupt_SCI6_TEI6(vect=VECT_SCI6_TEI6)
void Interrupt_SCI6_TEI6(void){} //disabilita trasmissione
```

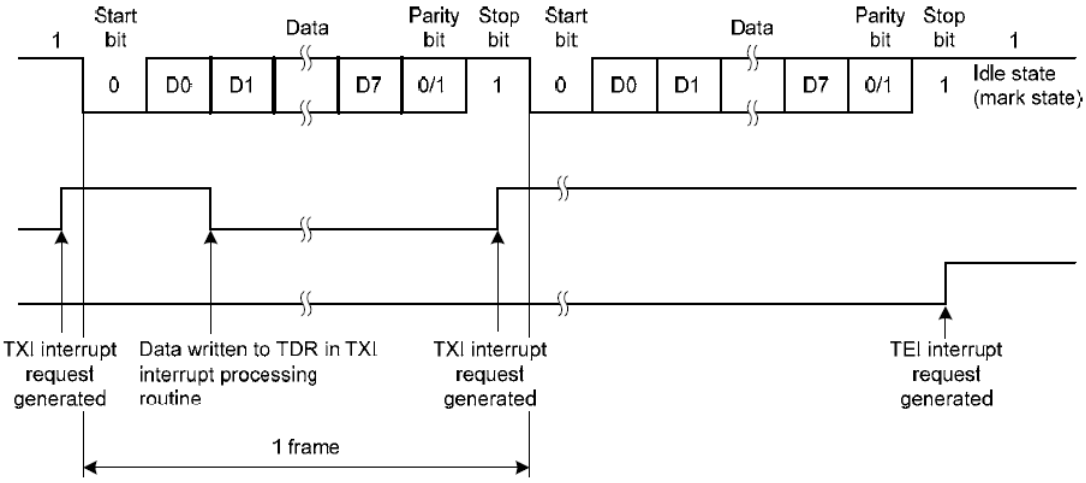


Figura 5.4: Tempistiche della generazione degli interrupt in trasmissione.

5.2.1 Gestione del break

Una parte importante del lavoro svolto è stata quella riguardante la *gestione del break*, in quanto si tratta di uno standard de facto in alcune applicazioni e per alcuni produttori[4] ma gestita solo in parte dalla nostra periferica. Si è dovuto fare in software quello che altri implementano direttamente in hardware.

Seguendo la descrizione presente nel *data sheet*, per mantenere la linea bassa bisogna disabilitare la trasmissione in modo da avere il controllo diretto del pin che viene impostato come uscita allo stato basso. Sfruttando il *loop*

[4]Come nel LIN Bus

back del transceiver CAN questo genera un errore sulla trama in ricezione gestito in interrupt alzando il pin e riabilitando la trasmissione. La durata del break in questo modo però non arriva agli 11 bit di specifica, che si sono raggiunti variando il registro BRR (solo durante questa procedura) fingendo quindi una frequenza minore.

Nel caso in cui la seriale sia configurata con otto bit di dati e uno di stop a 19200 baud la trama è composta in tutto da 10 bit, letti a metà del loro periodo. Per questo l'errore sulla trama viene generato mezzo bit prima del decimo. Riprendendo la formula **??** si calcola il nuovo BRR:

$$\frac{11 \cdot bit}{19200} = 572\mu s \qquad \frac{9.5 \cdot bit}{19200} = 494\mu s \qquad \frac{9.5 \cdot bit}{572\mu s} = 16608baud$$

$$BRR^* = \frac{48 \cdot 10^6}{32 \cdot 16608} - 1 = 90$$

Le schede collegate al bus sono sempre in ricezione e quando rilevano un break iniziano subito a campionare la linea nel RSR. Il protocollo Invece quì prevede una possibile pausa la cui gestione è stata introdotta disabilitando la ricezione fino al rilevamento dello start bit della prima trama. È stato evitato il *busy waiting* configurando una periferica MTU[5] a rilevare in interrupt il fronte di discesa.

Nel prima versione del microcontrollore con cui abbiamo lavorato l'abilitazione delle linee RX e TX non era indipendente ma per attivarne una si doveva prima spegnerle entrambe. In seguito alle nostre perplessità al riguardo espresse ai tecnici di riferimento (e con ogni probabilità non eravamo gli unici) hanno deciso di modificare i registri per slegarne la configurazione.

5.3 Driver per bus T4

Lo stack di protocolli che prende il nome di bus T4 si basano sul *layer fisico* realizzato con un driver che segue questa struttura. Si preoccupa solo di inviare sul bus (o di rilevare) un pacchetto di byte aggiungendo il controllo della lunghezza, di errore tramite un *checksum* e le tempistiche del break.

Bus T4 Il livello fisico per l'accesso al mezzo viene utilizzato da due *protocolli di rete*, uno si basa sull'indirizzo MAC univoco di ogni dispositivo mentre l'altro utilizza una struttura del tipo *indirizzo–porta* che identifica in maniera logica sia il dispositivo che l'applicazione a cui è destinato il pacchetto.

[5]Multi-Function Timer Pulse Unit

Questo ultimo viene utilizzato da due protocolli di livello *applicazione* che servono uno per inviare messaggi e l'altro per notificare eventi. Il primo viene impiegato nell'interrogazione e modifica dei parametri della centrale rendendo possibile la configurazione da remoto e prevede una apposita struttura per la gestione delle risposte. Il secondo segnala comandi rilevati dal modulo radio o generati da altri dispositivi.

Quello che usa il MAC serve e garantire una comunicazione univoca anche in presenza di due centrali con lo stesso indirizzo e ha lo scopo di facilitare la gestione della rete evitandone la configurazione in alcuni casi molto comuni.

A livello applicativo i pacchetti ricevuti vengono gestiti tramite delle *funzioni di call-back* richiamate in automatico dalla macchina a stati che gestisce la comunicazione ma lasciate in bianco, in maniera simile all'*implementazione di interfaccie* in Java e altri linguaggi ad alto livello.

I pacchetti vengono inviati secondo il loro valore di priorità che dipende dal tipo di protocollo che incapsulano e dal tempo di attesa, infatti se l'invio viene ritardato perché il bus è occupato da continue trasmissioni di pacchetti più importanti gli viene variata la priorità per consentirgli di essere comunque spedito.

Uno strumento molto utile in questo contesto è un piccolo palmare chiamato *oview* che ha una interfaccia per collegarsi al bus T4 e un software che permette di gestire la rete, interrogare e spedire comandi ai vari dispositivi. È stato molto utile anche nella *prima fase di collaudo* del driver e per verificare il corretto smistamento del blocco dati dei pacchetti secondo l'intestazione.

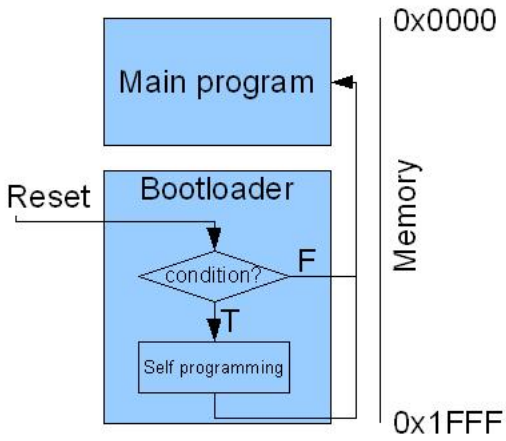


Figura 5.5: Modello di esecuzione di un bootloader.

Bootloader È prevista una estensione dello stack inserendo un nuovo protocollo dedicato all'aggiornamento del firmware interno alla centrale. Il software deve essere strutturato per *auto aggiornarsi* e questo è stato implementato utilizzando un *bootloader* come in figura 5.5, ovvero una piccola porzione di codice che viene eseguita a ogni avvio e normalmente ha il compito di passare il comando al programma principale ma nel caso rilevi una riprogrammazione gestisce il trasferimento dei dati e la loro scrittura nella parte di memoria destinata al programma.

Questo micro ha delle facilitazioni al riguardo riservando un'area di memoria dedicata e una modalità apposita dalla quale si ha accesso anche in scrittura alla flash riservata alle istruzioni.

Capitolo 6

Struttura del main

Finora si è descritto il funzionamento delle periferiche e si è avuto a che fare con i dettagli hardware del dispositivo. Adesso serve realizzare la logica dell'applicazione in un contesto più generale e ad alto livello.

All'accensione si esegue l'inizializzazione hardware della macchina, poi delle varie periferiche e si caricano le impostazioni iniziali. La centrale quando operativa non si comporta sempre allo stesso modo ma varia in base alla *macrofunzione* che sta svolgendo. Oltre a questo alcuni moduli devono rimanere sempre attivi e lavorare come se fossero dei processi indipendenti, ma senza avere a disposizione uno scheduler vero e proprio.

6.1 Inizializzazione hardware

È la prima cosa da fare quando parte il programma. Infatti non sono ancora definiti i valori che riguardano la gestione dell'oscillatore e dei clock sui vai bus (ICLK, PCLK, BCLK). Questo è anche il momento per verificare se si è appena eseguito un reset della macchina e dedurne dai flag il motivo in modo da gestire un problema rilevato dal WDT (sezione 7.2), il risveglio dalla più bassa modalità di risparmio energetico (sezione 7.1) oppure la volontà di entrare in modalità di aggiornamento del firmware tramite il bootloader (sezione 5.3).

Subito dopo si inizializzano tutte le periferiche che si intendono utilizzare. Dal momento che per default sono spente e ricordando la struttura modulare dell'applicazione questa parte viene realizzata richiamando una dopo l'altra tutte le routine di inizializzazione dei driver.

È importante rispettare le dipendenze tra le varie routine e la prima cosa in assoluto da configurare ed avviare è il tick di sistema (sezione 3.2.1) in modo da avere subito una temporizzazione sulla quale basare le operazioni

seguenti. Subito dopo si possono far partire led, switch, pwm, i2c, eeprom, sci facendo attenzione a far precedere l'eeprom dall'i2c su cui si basa.

Nella routine di configurazione di molte periferiche si devono specificare le impostazioni per selezionare il clock adatto alle proprie esigenze derivandolo da quello principale. Spesso quest'ultimo non è fisso ma viene cambiato durante lo sviluppo della stessa applicazione al variare delle esigenze o nella realizzazione di altri progetti. Per questo i valori da inserire nei registri non sono fissi ma vengono calcolati dinamicamente in base al valore del clock principale usato e al valore di uscita voluto. Le espressioni matematiche utilizzano come parametri valori costanti specificati nei *define* dei file *header* e per questo motivo quando il compilatore le tratta le sostituisce subito con il risultato e questo quindi non pesa sul programma finale.

```
#include "led.h"
#include "switch.h"
..
void main()
{
  Timer0_Init(); //timer per base tempi
  Led_Init();  //led
  Switch_Init();  //switch
  ..
  Param_Read_EE();
  t4_init();
  ..
  while(1)
  {
    //task led
    //task pulsanti
    //task bus t4

    //macchina a stati
    if(stato==0){}
    else if(stato==1){}
    else if(stato==2){}
    ..
  }
}
```

6.2 Inizializzazione software

Una volta che tutto l'hardware è disponibile si inizializzano i parametri di configurazione caricandoli dall'eeprom sfruttando le routine sviluppate appositamente (sezione 4.3). Ora si hanno anche le informazioni sulla rete e i protocolli per avviare il bus t4 (sezione 5.3), ma che rimane ancora non disponibile in questo punto perché i task non sono ancora attivi.

La fase successiva prevede la ricerca delle fotocellule su *BlueBus*[1] che è una operazione che richiede alcuni secondi durante i quali viene visualizzato un particolare lampeggio nei led e devono essere disponibili tutte le risorse dell'applicazione. Per questo motivo questa fase, pur appartenendo logicamente all'inizializzazione, non rientra tre le operazioni da eseguire subito ma è stata spostata come macrofunzione all'interno della macchina a stati. Si è così anche guadagnato il fatto che è possibile rieseguirla in un secondo momento.

6.3 Macchina a stati

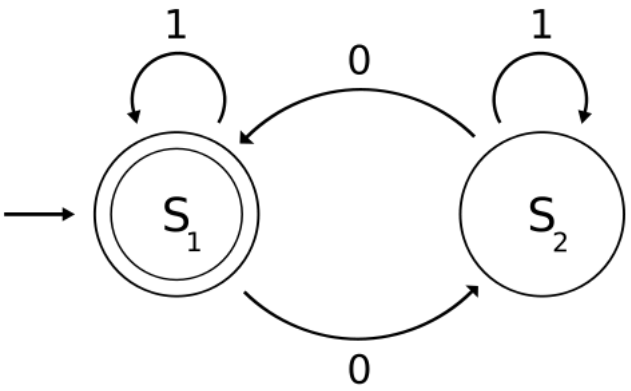


Figura 6.1: Esempio di automa a stati finiti.

Una macchina a stati finiti è un sistema dinamico che può assumere solo un determinato numero di stati fra i quali si muove. Ingressi e uscite sono discreti e il comportamento è ben definito e sempre uguale a fronte delle stesse condizioni. Anche il tempo scandito dal tick di sistema rappresenta un ingresso come gli altri.

Questo è un modo di vedere semplice ed efficace facilmente implementabile e che richiede poche risorse. Può essere realizzato inserendo in un ciclo infini-

[1] Non implementata

to una struttura del tipo *switch-case* che lavora su una variabile che rappresenta lo stato. Ogni ramo dello switch deve avere l'importante caratteristica di essere *non bloccante* ovvero di venire eseguito in maniera lineare senza cicli dalla durata indefinita o *busy waiting*. Questa limitazione viene superata sfruttando il ciclo esterno che riesegue lo stesso codice finché lo stato non cambia e utilizzando ulteriori variabili di stato interne.

Nella nostra applicazioni si sono identificate alcune macrofunzioni mutuamente esclusive descritte brevemente di seguito:

stabile In attesa di comandi.

acquisizione dispositivi Cerca fotocellule sul BlueBus.

acquisizione quote Salva quote da muovi comunque.

cancellazione memoria Carica le impostazioni di default.

apertura Manovra di apertura.

chiusura Manovra di chiusura.

fermata Arresto motore e diagnostica.

programmazione Modifica parametri regolabili.

muovi comunque Manovra manualmente.

La struttura è molto versatile e può essere facilmente modificata introducento una parte che viene eseguita a ogni iterazione indipendentemente dallo stato effettivo. Anche questa deve avere la caratteristica di essere non bloccante ed esiste quindi una durata massima, nel caso peggiore, di esecuzione di una iterazione. Se la frequenza con cui si susseguono i cicli è più veloce delle noste esigenze si riesce a simulare la contemporaneità delle operazioni.

Questa è la posizione adatta ad inserire i *task* per la gestione dei vari processi come ad esempio l'aggiornamento dei led, la lettura degli switch, la gestione dei bus ...

Se sono particolarmente sensibili alle latenze si possono far eseguire ad ogni iterazione[2] altrimenti possono venire *rallentati* in modo da evitare continue operazioni inutili e guadagnare in prestazioni. Come accennato nel paragrafo 3.2.1 si può usare il *tick timer* sfruttando le funzioni *tick_current* e *tick_elapsed* per richiamare il task solamente ad intervalli regolari e indipendenti secondo le esigenze creando così la possibilità di specificare le *priorità*.

[2]Come il bus T4

Il passaggio da uno stato all'altro viene effettuato sia dallo stato attivo che da un task.

Le risorse comuni vengono assegnate utilizzando dei semafori in modo da riservarne l'accesso ad un solo utilizzatore alla volta (mutua esclusione) e anche per *sincronizzare* i vari task.

Capitolo 7

Risparmio energetico

Attualmente c'è molta attenzione rivolta al risparmio energetico e al basso consumo, soprattutto quando in stand by. Se un dispositivo è fermo non dovrebbe consumare energia. Questo genere di applicazione però non può rimanere spento del tutto: il modulo radio, i controlli sulle fotocellule e altre sicurezze devono essere sempre attivi per poter rispondere agli eventi esterni.

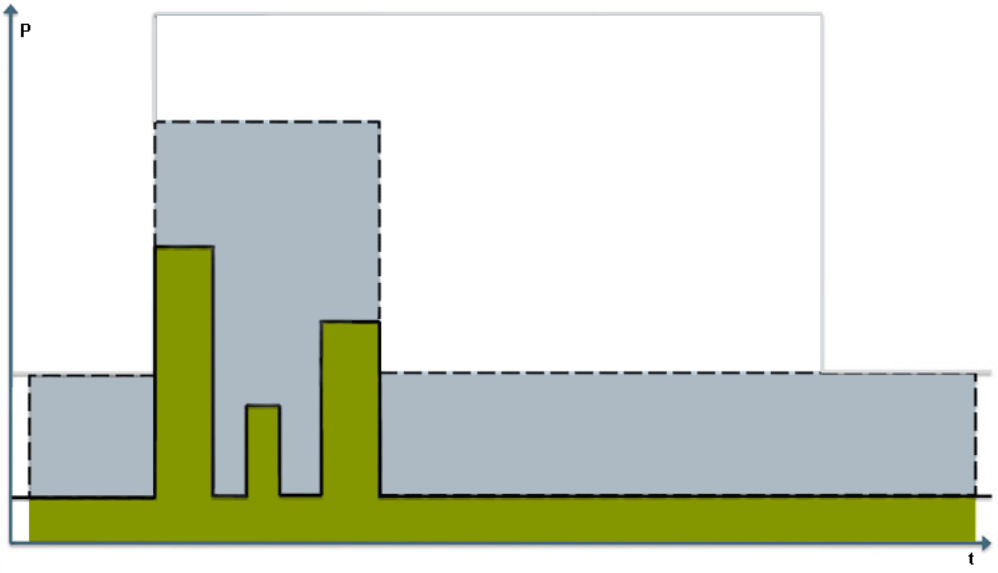


Figura 7.1: Principi del risparmio energetico.

Si agisce quindi lavorando su due fattori distinti, come in figura 7.1, diminuendo la potenza assorbita e riducendo il tempo di funzionamento. La struttura modulare delle periferiche permette di tenere accese solamente quelle

che servono disabilitando le altre e abbassando l'assorbimento generale di corrente.

$$P = V \cdot I \qquad L = P \cdot t$$

Anche la frequenza di lavoro influisce sul consumo secondo la formula **??**[1] ed è quindi opportuno sceglierla in base alle proprie necessità. Se troppo alta si sprecano istruzioni ed energia altrimenti si rischia di non riuscire a svolgere correttamente il proprio compito rispettando le tempistiche richieste.

$$500\mu A/MHz \tag{7.1}$$

L'architettura a molti bus (vedi figura A.1 in appendice) rende possibile usare una frequenza diversa e ottimale per ognuno favorendo il core, le periferiche ed eventuali comunicazione esterne[2]. Questo è stato ottenuto derivando i vari clock da quello principale attraverso dei divisori di frequenza. Spesso non si sfruttano tutte le potenzialità del micro e alcune impostazioni risultano indifferenti ai fini dell'applicazione ma non dal punto di vista dei consumi. Infatti impostare una frequenza derivata e non usata al valore minimo può sembrare la scelta migliore ma spesso non lo è perché questo fa caricare i flip-flop dei divisori di frequenza che consumeranno di più.

I maggiori risultati però si ottengono sospendendo l'esecuzione del firmware in tutti quei momenti in cui non è strettamente necessario. Ad esempio se un sistema deve monitorare una temperatura spesso non serve leggere continuamente il sensore ma basta farlo a intervalli sufficientemente brevi da garantirne il corretto funzionamento. Così se basta una lettura al secondo della ipotetica durata di $10ms$ si riduce l'attività del 99%.

Nel caso si richieda anche di elaborare quel dato si aggiunge il tempo necessario a farlo, che dipende dalla velocità del micro. Questo potenzialmente significa che più il dispositivo è prestante meno ci mette a svolgere il suo compito e quindi può rimanere più tempo nella modalità di basso consumo. Secondo questa logica e in questi casi una architettura a 32 bit che richiede mediamente più potenza di una ad 8 bit può consumare invece complessivamente di meno.

7.1 Modalità di risparmio

I microcontrollori hanno questa potenzialità che in alcuni è particolarmente sviluppata rendendo possibili diverse modalità di risparmio energetico ottenute spegnendo progressivamente vari blocchi. Si avranno disponibili sola-

[1] Riferita alla serie RX600, con tutte le periferiche attive

[2] Come SRAM o display.

mente alcuni sottoinsiemi di funzioni e quindi anche di interrupt che possono risvegliare il micro.

Ci sono quattro modalità di basso consumo diverse:

Sleep Solamente il core è fermo.

All-module clock stop Vengono fermate anche tutte le periferiche (tranne TMR, WDT, IWDT, RTC, power-on reset, voltage detection).

Software standby Viene fermato l'oscillatore (rimane: voltage monitoring, RTC).

Deep software standby Viene tolta l'alimentazione ai registri del core e delle periferiche ed a uno o entrambi i banchi di RAM.

Negli ultimi due modi l'oscillatore viene fermato eliminando uno dei componenti più esigenti in termini energetici ma introducendo il notevole svantaggio di doverlo riavviare al risveglio, rallentando il tempo di risposta e rendendole utilizzabili solamente nei casi di rare richieste e tollerando alte[3] latenze.

Nella figura 7.2 si possono paragonore meglio le varie risorse disponibili in funzione della modalità di basso consumo.

Operation	Sleep Mode	All-Module Clock Stop Mode	Software Standby Mode	Deep Software Standby Mode
State after cancellation	Program execution (interrupt processing)	Program execution state (interrupt processing)	Program execution (interrupt processing)	Program execution (reset processing)
Oscillator	Operating	Operating	Stopped	Stopped
CPU	Stopped	Stopped	Stopped	Stopped / Undefined
On-chip RAM 1	Operating	Retained	Retained	Undefined
On-chip RAM 0	Operating	Retained	Retained	Retained/ Undefined
USB2.0 host/function module	Operating	Stopped	Stopped	Retained/ Undefined
Watchdog timer (WDT)	Operating	Operating	Retained	Undefined
Independent watchdog timer	Operating	Operating	Retained	Undefined
8-bit timer (unit 0, unit 1)	Operating	Operating	Retained	Undefined
Peripheral modules	Operating	Stopped	Stopped	Undefined
I/O pin state	Operating	Retained	Retained	Retained

Figura 7.2: Modalità e risorse disponibili.

Il *Real Time Clock Calendar* (RTC) può generalmente dare interrupt a intervalli periodici, al variare di una cifra (carry) oppure al raggiungimento di

[3] Nell'ordine di $10ms$

un *timestamp* prestabilito, ma solo quest'ultimo funziona quando l'oscillatore è fermo. Nelle ultime due modalità infatti gli unici altri eventi interni che possono risvegliare il micro sono quelli dell'USB o un abbassamento della tensione di alimentazione, condizioni molto restrittive che rendono *necessario affidarsi a segnali esterni*, se disponibili.

In figura 7.3 si osserva la dinamica del passaggio dall'esecuzione attiva del programma al basso consumo, ricorrendo alla chiamata dell'istruzione *WAIT*.

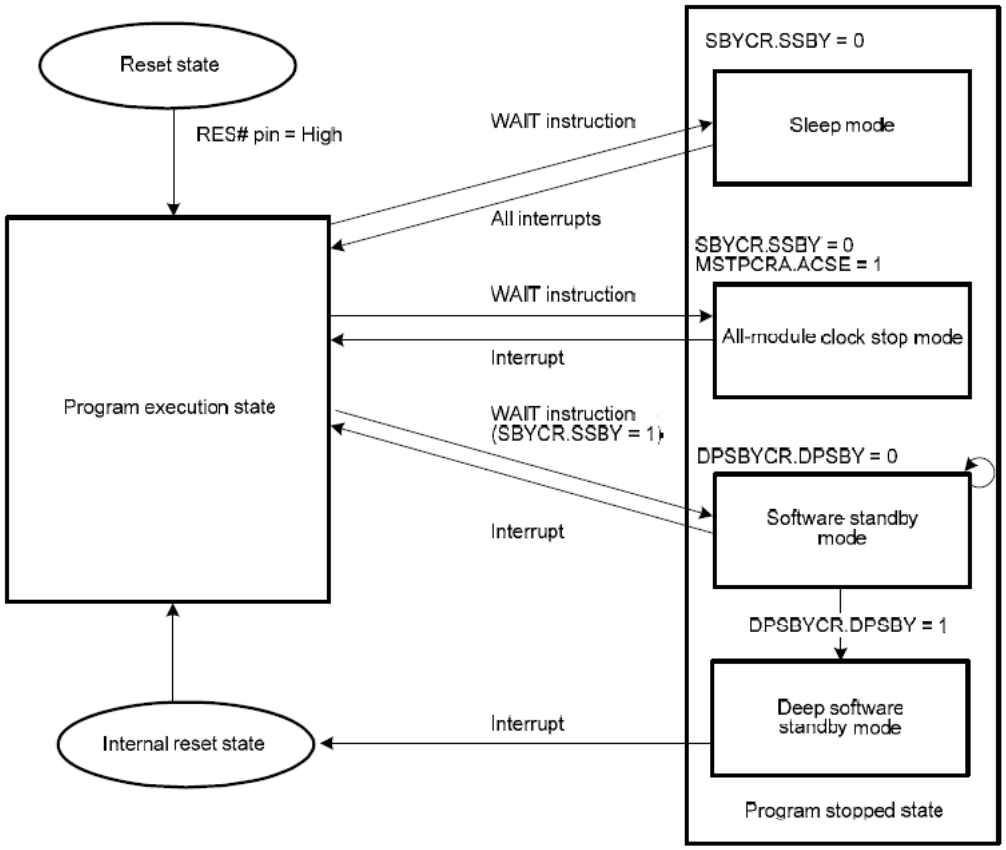


Figura 7.3: Schema di transizione tra modalità di risparmio energetico.

Questa è un'istruzione *privilegiata* che può essere eseguita solo in *modalità supervisore*. Per passare dalla modalità utente a quelle supervisore si deve generare una *eccezione*[4]. L'esecuzione dell'istruzione "brk" (break) in modalità utente genera una eccezione, interrompendo il flusso normale dell'esecuzione e portando il processore in modalità supervisore. Adesso si può eseguire

[4]Simile a un interrupt

il "wait" che porterà alla modalità di risparmio energetico selezionata (impostata dai registri, come in figura 7.3), che sarà interrotta da uno degli eventi precedentemente abilitati (interrupt vari).

Prima dell'esecuzione del "wait" ci si deve accertare che sia stata completata l'ultima scrittura su registro, che potrebbe essere posticipata al risveglio. La ISR dell'interrupt risvegliante viene eseguita prima che il processore esca dalla modalità supervisore.

Al termine dell'ISR l'esecuzione continua riprendendo dalla routine di gestione dell'eccezione e al termine di questa si ritorna in modalità utente e si riprende l'esecuzione normale del flusso del programma dall'istruzione successiva al brk.

Un altro modo per entrare in modalità supervisore consiste nell'esecuzione di una istruzione privilegiata (es. wait stessa) in modalità utente ma al ritorno dalla routine che gestisce l'eccezione ciò comporterebbe la ri-esecuzione dell'ultima istruzione, generando nuovamente l'eccezione. Pertanto questo metodo non è utilizzabile.

7.1.1 Misure sperimentali

Per verificare il funzionamento di questa parte dell'applicazione si è realizzato un progetto dedicato che gira sulla demoboard senza l'ausilio del debugger, in modo da non alterare il carico del micro. Si è deciso anche di non utilizzare led o carichi variabili e per monitorare lo stato del programma si è usato il display LCD presente nella scheda, comandato in parallelo e con capacità di memoria che non assorbe corrente dalle linee col micro per mantenere l'informazione.

Il programma è molto semplice e visualizza sul display l'ora attuale usando l'RTC sia quando attivo che in sleep, sfruttando gli interrupt periodici. Alla pressione di un tasto[5] si passa alla modalità più bassa evidenziando il restringimento dell'insieme di eventi disponibili per il risveglio, perdendo prima il periodico dall'RTC e poi, in *Deep software standby*, anche lo stesso tasto perché associato a una linea troppo alta, rimangono infatti solo da IRQ_0 a IRQ_3, verificato aggiungendo un pulsante in IRQ_2.

Come espresso anche nella figura 7.3 una volta in *deep* non è più possibile risvegliarsi nel senso convenzionale ma si esegue un reset della macchina, avendo così la possibilità di ri-inizializzare registri e periferiche andati persi. All'accensione si può comunque distinguere questo da un altro tipo di reset analizzando il valore di alcuni flag e discriminando anche il specifico tipo di interrupt di risveglio.

[5] Associato alla linea IRQ_9 di interrupt esterni

Le due demoboard non sono identiche ma appartengono a due versioni successive. La prima, più un prototipo, dispone di un *jumper* che interrompe l'alimentazione diretta al micro con lo scopo di misurarne l'assorbimento ma non sono state raggruppate tutte le linee di alimentazione rendendo impossibile una misura diretta. Nella seconda versione, avuta per debuggare la seriale, questo problema è stato risolto modificando leggermente lo stampato.

Togliendo il jumper e inserendo un multimetro[6] in serie configurato come amperometro si sono eseguite delle misure di corrente per confermare la correttezza del codice confrontando i valori ottenuti da quelli proclamanti nella parte elettrica del datasheet (figura in appendice A.3) che per stimare il consumo futuro nell'applicazione.

Active	Sleep	All-Stop	Software Standby	Deep Standby
$28.46mA$	$15.40mA$	$12.52mA$	$290\mu A$	$90\mu A$
100%	54.11%	43.99%	1.01%	0.32%

Tabella 7.1: Misure di assorbimento (valori medi).

Si sono fatte varie prove nella configurazione delle periferiche e nella gestione del clock al fine di minimizzare il consumo energetico.

Riprogrammando la demo con il firmware dell'applicazione finora sviluppata si è visto che assorbe circa $42mA$ dovuti all'utilizzo di più periferiche. Si ipotizza che nella scheda elettrica che si sta sviluppando il consumo del micro sarà minore perché ci saranno meno reti elettriche collegate ai vari pin e che richiedono meno corrente. Inoltre verrà scelto un componente con un minore taglio di memoria.

7.2 WDT e IWDT

Un *watchdog timer* (letteralmente "cane da guardia") è un meccanismo di sicurezza che ha il compito di resettare la macchina nel caso di cicli infiniti o quando il sistema non risponde. Il meccanismo è semplice ma efficace, consiste nel programmare un apposito timer a generare il reset entro un certo intervallo di tempo e nell'inserire in appositi punti nel programma l'azzeramento di quel timer (vedi figura 7.4). Se tutto va bene il timer viene regolarmente scritto (feed/pet/kick the dog) e non arriverà mai a raggiungere il valore finale nel quale resetta il micro.

[6]Hewlett Packard 34401A, $6^{1/2}$ cifre di risoluzione

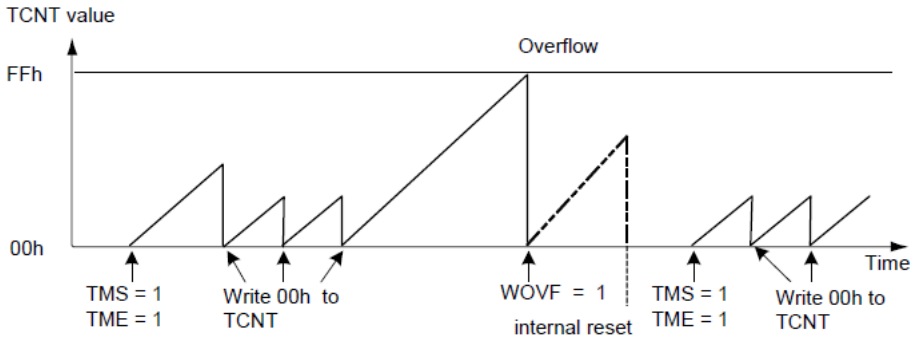


Figura 7.4: Esempio operativo del WDT.

Per rispondere ai requisiti di sicurezza del software un WDT normale che deriva la propria frequenza dal clock di sistema non è adatto. Per questo è stato introdotto il WDT *indipendente*, che utilizza cioè un proprio oscillatore a bassa frequenza interno al micro. L'IWDT introduce un'altra novità molto importante che rende possibile l'accesso al contatore solamente entro certe *finestre temporali* impostabili, in questo modo l'azzeramento del timer deve si avvenire entro una certa soglia ma non troppo presto, aumentandone l'efficacia e la robustezza dell'applicazione.

Al reset si entra in una routine di messa in sicurezza delle uscite, fermando l'automatismo in modo che non possa arrecare danni a cose o persone.

Quando un WDT è operativo l'esecuzione dell'istruzione "wait" *non può portare allo standby* del micro[7] ma si andrà in *sleep* oppure in *all module clock stop*.

[7] sia software che deep

Conclusioni

Il progetto è ampio e lungo ad essere completato ma si vedono già i primi risultati. Con il firmware prodotto fino ad ora la demoboard è già in grado di effettuare alcuni dei compiti richiesti all'applicazione e di interfacciarsi con altri prodotti utilizzando il bus T4.

La prossima cosa da sviluppare è la gestione del protocollo BlueBus facendo il porting del relativo codice e scrivendo il driver per il convertitore analogico/digitale che sfrutti le potenzialità della periferica e soprattutto quelle offerte dall'architettura attraverso il DTC e il DMA, strumenti che permettono di automatizzare campionature successive senza basarsi sugli interrupt e caricare la CPU.

Molto istruttivo è stato lavorare allo stesso progetto con altre persone, capire le necessità, dividersi i compiti, interfacciarsi con i rappresentanti per conoscere nuovi prodotti e possibilità. Prendere in mano il codice di altre persone ti insegna nuove accortezze implementative e a organizzare il tuo in modo che sia facile da leggere, capire e modificare.

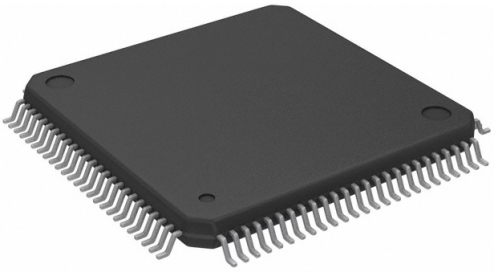
La diffusione di questa struttura è limitata dalla maggiore occupazione di memoria dovuta all'introduzione dello strato che svincola l'hardware dal software e all'adozione di un linguaggio ad alto livello, anche se ottimizzato. I vantaggi che si hanno tuttavia sono notevoli e irrinunciabili per lo sviluppo futuro.

La macchina a stati deve essere eseguita velocemente per servire entro i limiti temporali tutti i task i quali devono essere scritti in modo non bloccante e servendosi dei semafori per l'utilizzo di risorse condivise e per la sincronizzazione. Questo fa si che la scrittura del codice deve essere accorta e tenere sempre in mente che in un qualsiasi punto il programma può essere interrotto da un interrupt che potrebbe alterare lo stato delle variabili comuni.

L'evoluzione futura porta verso l'introduzione di un sistema operativo real time (RTOS) per la gestione dei task, il supporto ai semafori e all'allocazione dinamica della memoria.

Appendice A

Micro



specifica	FREESCALE	MICROCHIP	NXP	RENESAS	STM	TI
Flash (KB)	256	256	512	384	256	256
ROM (KB)	–	–	8	Y	–	Y
RAM (KB)	32	32	64	64	64	96
Frequency (MHz)	66	72	100	100	72	80
Perf. (MIPS/MHz)	0.98	1.56	1.25	1.65	1.25	1.25
Internal oscillator	Y	Y	Y	–	Y	Y
I/O	58	82	70	74	80	65
Temp Range	-40 +85	-40 +85	-40 +85	-40 +85	-40 +105	-40 +85
FPU	–	–	–	Y	–	–
DSP aritm. func.	Y	Y	–	Y	–	–
CRC calculator	–	Y	Y	Y	Y	Y
Watchdog	2 ind.	1	1	2 ind.	2 ind.	2 ind.
Timer 32bit	4	2	4	2	S	–
Timer 24bit	–	–	–	–	S	1
Timer 16bit	2	5	–	16	4	4
Timer 8bit	1	–	–	4	S	–
RTC	Y	Y	Y	Y	Y	Y
Ethernet	1	1	1	Y , DMA	1	Y
CAN	1	1	2	1	2	2
USB (H, D, OTG)	1	1	1	1	1	1
UART	3	2	4	6	5	3
I2C	2	2	3	1/2	2	2
SSI/SPI	1	2	3	2	3	2
I2S	–	–	1	–	3	1

Tabella A.1: Comparativa (*continua...*)

specifica	FREESCALE	MICROCHIP	NXP	RENESAS	STM	TI
LIN	–	Y	–	–	Y	–
ADC indip (n/bit)	1/12	1/10	1/12	1/12, 2/10	2/12	2/10
ADC ch tot	8	16	8	16	16	16
ADC speed (MHz)	1	0.4	0.2, 12bit	1	1	1
Temp sens inside	–	–	–	–	Y	Y
DAC (n/bit)	–	Y	1/10	1/10	2/12	–
ANALOG COMP	–	2	N	–	S	3
DIGITAL COMP	–	–	N	–	S	16
External bus	Y	Y	–	Y	S	Y
PWM	8	5	6	36	6	8
CCP	6	5	N	21	S	8
QEI ch	–	–	1	2	4	2
Stand by	Y	Y	Y	Y	Y	Y
DMA/DTC	4	4	Y	Y	Y	Y
Eeprom/DFlash	Y	Y	64K DF	32K DF	S	Only DF
JTAG	Y	Y	Y	Y	Y	Y
On chip debugger	Y	Y	Y	Y	Y	Y
Flash HW protect.	Y	Y	Y	Y	S	Y
5V tollerant pin	–	Y	Y	7	Y	0-67
OP amp inside	–	–	N	–	–	–
Class B complain	Y	Y	Y	Y	Y	Y
Criptatura	Acc.	–	–	–	–	Y, hw
RND generator	Y	–	–	–	–	–

Tabella A.2: Comparativa.

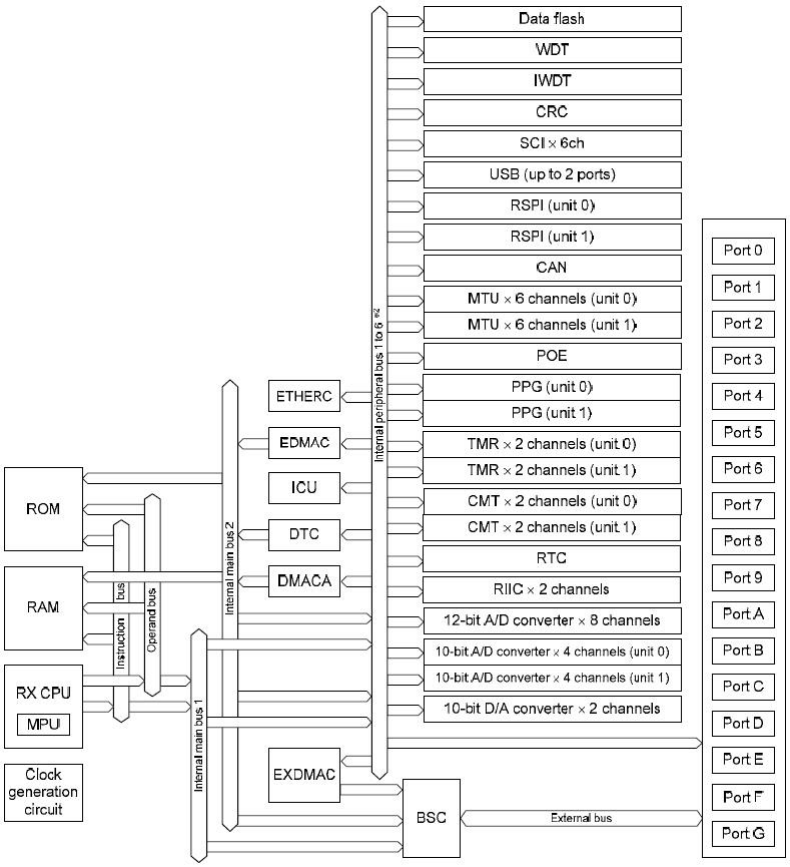


Figura A.1: Schema a blocchi, bus e periferiche

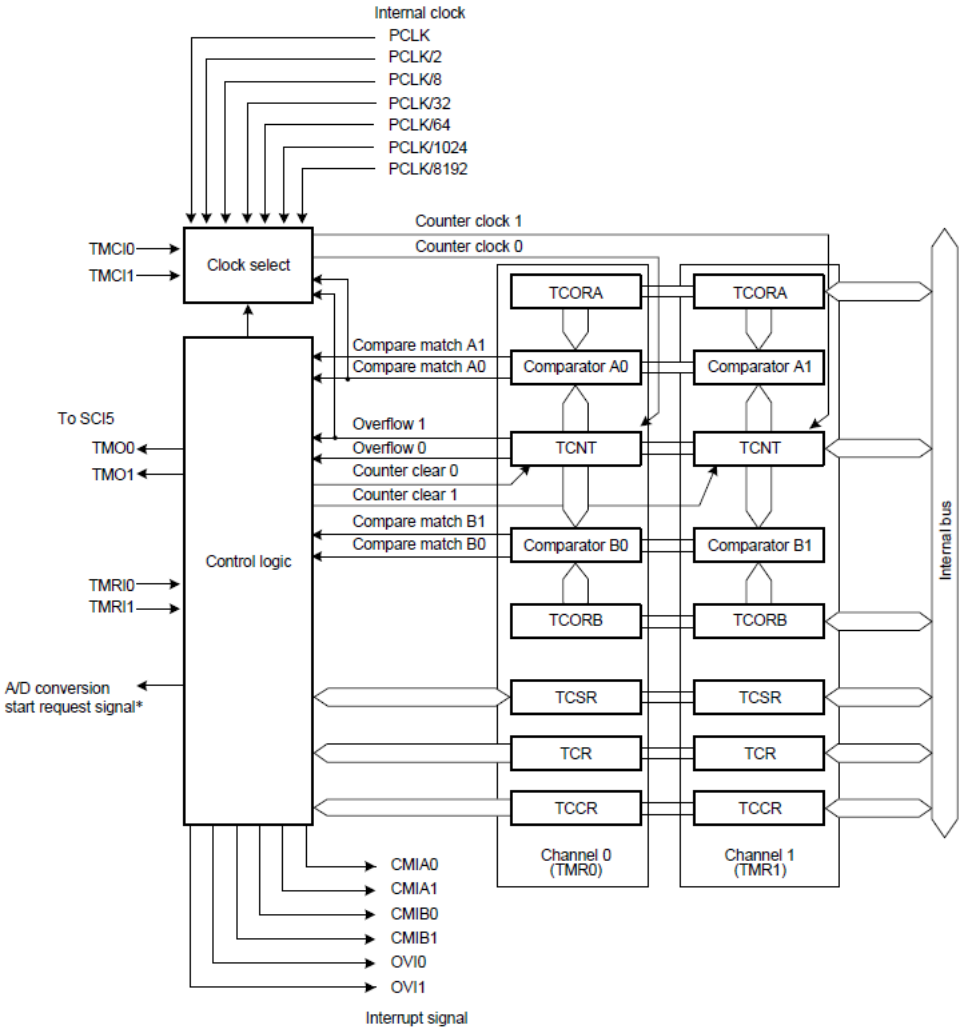


Figura A.2: Struttura dei timer a 8 bit

Item					Symbol	Min.	Typ.	Max.	Unit
Supply current	In operation	Max.			I_{CC}****	-	-	100	mA
		Normal				-	35	-	
		Increased by BGO operation				-	15	-	
	Sleep					-	20	60	
	All-module-clock-stop mode					-	14	28	
	Standby mode	Software standby mode				-	0.12	3.0	mA
		Deep software standby mode	RTC in operation	RAM, USB retained		-	30	206	µA
				RAM, USB power supply halted		-	26	66	µA
			RTC halted	RAM, USB retained		-	25	200	µA
				RAM, USB power supply halted		-	21	60	µA

Figura A.3: Consumo del micro secondo la modalità di risparmio energetico.

Appendice B

Demo

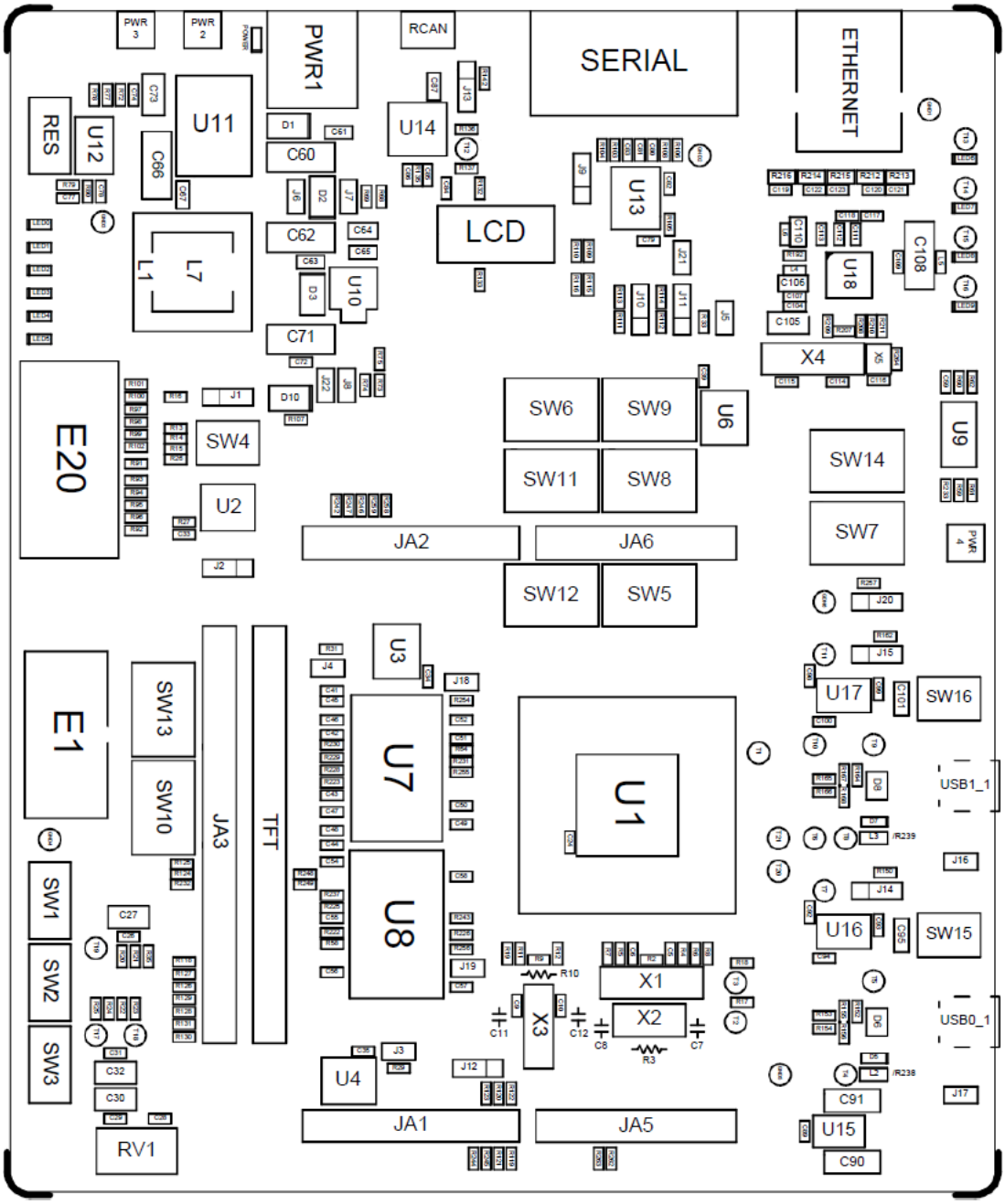


Figura B.1: Demo: Top Layer.

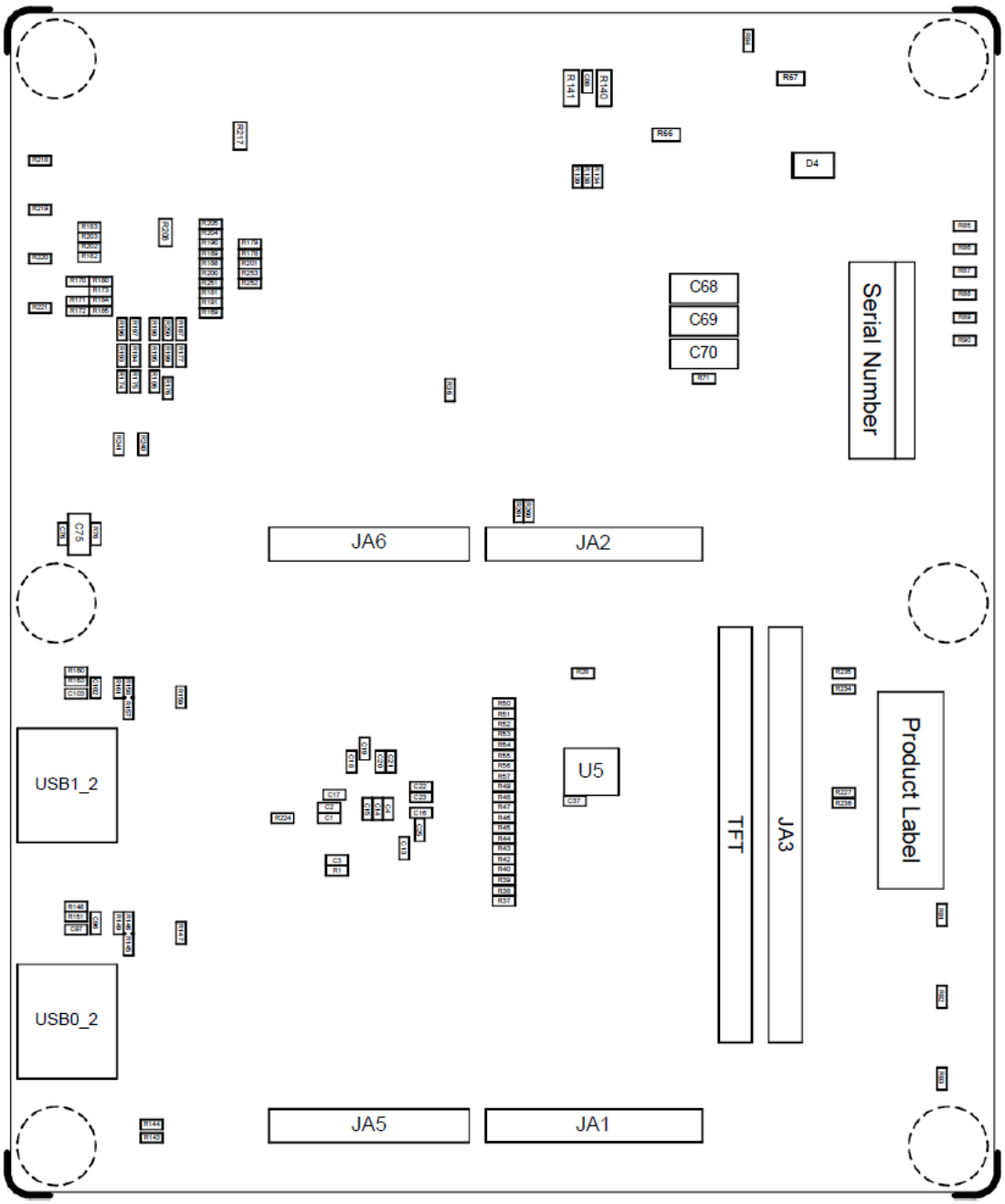


Figura B.2: Demo: Bottom Layer.

Bibliografia

[1] http://www.freescale.com/webapp/sps/site/homepage.jsp?code=PC68KCF

[2] http://www.microchip.com/en_US/family/pic32

[3] http://www.nxp.com/#/pip/pip=[pfp=56890]|pp=[t=pfp,i=56890]

[4] http://www.rxmcu.com/USA/core.php

[5] http://www.st.com/internet/mcu/subclass/1169.jsp

[6] http://focus.ti.com/mcu/docs/mculuminaryprodoverview.tsp?sectionId=95&tabId=2486&familyId=1755&DCMP=Luminary&HQS=Other+OT+stellaris

[7] http://en.wikipedia.org/wiki/I2C