# Relazioni Laboratorio Elettronica

Giorgio Pizzati

Novembre 2019

# Indice

# Capitolo 1

# Microcontrollore 8051

## 1.1 Introduzione al microcontrollore 8051

Il microcontrollore 8051, il primo dei due usati in questo corso ha una struttura di tipo Harvard(come tutti i microcontrollori).

- memoria programmi distinta memoria dati, alcuni anche doppio bus
- 8bit
- interrupt non nested
- assembly

### 1.1.1 Configurazione del micro

Qui di seguito viene riportata la configurazione generica eseguita all'inizio di ogni esperienza con l'8051.

Il primo passo per configurare il microcontrollore è disattivare il watchdog. Per fare ciò viene usato il Configuration Wizard. All'interno di questo software viene creato un file assembly che viene incluso nel progetto del'8051 ed eseguito all'inizio del main mettendo in funzione tutte le periferiche interessate. Un ulteriore step da fare in fase di configurazione è quello di indicare al micro una sorgente esterna per il clock. La sorgente che useremo è un cristallo oscillatore che ha una frequenza di 22.1184 MHz. Questo passaggio è necessario per garantire al micro di avere una frequenza di clock più precisa in quanto quella interna è di $(2 \pm 0.02)$MHz. Una volta completata la configurazione nel Wizard, si esporta un file .asm che viene incluso nel progetto e nel file contenente il programma(e.g."led.a51") viene dichiarata una funzione esterna Init_Device che fa riferimento a quella scritta dal Wizard.

## 1.2 Relazione led

### 1.2.1 Led in assembly in polling

In questa esperienza si è programmato il microcontrollore per accende e spegnere il led. Il codice è stato scritto in assembly utilizzando il timer in polling.

Per la realizzazione è necessario impostare la porta P1.6 che controlla l'accensione e lo spegnimento del led, in configurazione push-pull ed impostare il Timer0. Per realizzare i due cicli su due registri distinti sono stati usati R0 per i cicli che durano un overflow del timer ed R1 per i 10 cicli in cui viene acceso e spento il led. Dopo aver configurato il micro, nel main, inizia un loop che dura 10 cicli, all'interno dei quali:

- si accende il led
- si chiama la funzione timer, all'interno della quale il micro starà per un tempo $\tau$
- si spegne il led
- si chiama di nuovo timer
- si ritorna all'inizio del loop

Listing 1.1: Main del programma

```
MOV R0, #10
REPEAT:
SETB P1.6 ;accendo il led
LCALL TIMER
CLR P1.6 ;spengo il led
LCALL TIMER
DJNZ R0, REPEAT
```

La funzione timer è implementata come una serie di cicli(50), durante i quali:

- si aspetta un overflow del timer in un while loop
- si resettano le variabili del timer
- si controlla se sono passati 50 cicli

Listing 1.2: Funzione timer

```
TIMER:
  ;faccio trascorrere un po' di tempo
  MOV R1, #50 ;voglio che venga eseguito due volte
  FOR:
  SETB TR0 ;faccio partire il cronometro
  WHILE:
```

```
    JNB TF0, WHILE ;
  CLR TR0 ; FERMO IL TIMER
  CLR TF0 ;AZZERO IL FLAG DI OVERFLOW
  SETB TR0 ; faccio ripartire il timer
  DJNZ R1, FOR ; loop eseguito 50 volte?
  CLR TR0; fermo il timer
  CLR TF0 ;azzero il flag di overflow
  RET
```

### 1.2.2 Led in assembly con interrupt

La gestione dell'interrupt è il fulcro di questa esperienza. Innanzitutto quindi bisogna abilitarlo nel configuration wizard. È stato poi necessario capire come gestirlo. È stato necessario fare un code segment a 0x000B dove punta l'interrupt del timer0, ovvero la porzione di codice che viene eseguita quando scatta l'interrupt. Visto che lo spazio riservato in 0x000B permette di eseguire solo un'istruzione è necessario fare un JMP ad un'area di memoria programmi in cui è possibile svolgere più istruzioni. In quest'area viene gestito completamente l'interrupt dalle funzioni sotto elencate.

Listing 1.3: Funzione timer

```
TIMER:
    CLR TR0; viene fermato il timer
    CLR TF0; si pulisce la flag di overflow
    DJNZ R0, NULLA; si decrementa R0, salta se=0
    JMP SCELTA; questo jump viene eseguito se R0=0

NULLA:
    SETB TR0; faccio ripartire il micro
    RETI; si esce dall'interrupt
```

Scelta è una semplice funzione che determina cosa fare, se spegnere oppure accendere il led.

Listing 1.4: Funzione scelta

```
 SCELTA:
  ;SE R1 = 1 ALLORA ACCENDO, SE = 0 SPENGO
  MOV A, R1; si sposta nell'acc. R1 per fare l'if
  JZ ACCENSIONE
  JNZ SPEGNIMENTO
```

Mentre le funzioni accensione e spegnimento sono le seguenti

Listing 1.5: Funzioni accensione e spegnimento

```
ACCENSIONE:
  SETB P1.6; si accende il led
```

```
  MOV R0, #255 ; si imposta R0 a 255
  MOV R1, #1 ; si imposta lo status a R1
  SETB TR0; faccio ripartire il micro
  RETI

SPEGNIMENTO:
  CLR P1.6; si spegne il led
  MOV R0, #255
  MOV R1, #0
  SETB TR0; faccio ripartire il micro
  RETI
```

R1 quindi in questo caso è un registro di stato, tiene in memoria lo stato del led, se è acceso o spento.

Ciò che viene eseguito nel main è solamente la configurazione del micro, viene messo in R0 l'indirizzo di 255 e in R1 quello di 0.

### 1.2.3 Led in C

Scopo di quest'esperienza è di apprendere l'interazione tra il programma in C e quello in assembly. La parte di configurazione del micro infatti rimane scritta in asm mentre il codice principale è scritto in C. L'interazione in realtà si limita a dichiarare nel file .c una funzione esterna Init Device definita nel file .asm. Il programma in questo caso, invece di limitarsi ad accendere e spegnere il led, permette di regolarne la luminosità tramite il bottone disposto sulla scheda di sviluppo. Dunque le novità del codice rispetto a quelli precedenti sono

- codice in C
- gestione della luminosità del led
- gesitone dell'interrupt del bottone

Quindi andando in ordine il codice in C: volendo operare solo su alcuni bit dei registri di controllo è stato necessario definirli a partire dalle variabili dichiarate nel file C8051f020.h in questo modo:

```
sbit P1_6 = P1 ^ 6;
sbit P3_7 = P3 ^ 7;
```

che permette di selezionare solo i bit che ci interessano.

Per quanto riguarda la luminosità, che in realtà non è regolabile sul micro, viene usato un'escamotage modificando la frequenza con cui viene acceso e spento il led. All'occhio umano il cambio di frequenza di accensione appare come una differente luminosità in quanto la frequenza è paragonabile con il refresh rate dell'occhio.

La gestione dell'interrupt del bottone viene eseguita in modo molto simile a quella del led, tranne per il fatto che il flag dell'interrupt non è bit addressable

quindi per resettarlo è necessario usare una maschera e per il numero di interrupt assegnato. Come ulteriore differenza dall'asm infatti si ha che per gestire l'interrupt è necessario definire funzioni il cui nome è seguito da "interrupt" e da un numero che identifica quale interrupt si sta gestendo.

Loops identifica quanti cicli di overflow del timer si aspettano prima di cambiare lo stato del led, ovvero identifica la frequenza di accensione e spegnimento. Questo controllo viene effettuato all'interno dell'interrupt del timer.

Listing 1.6: Interrupt timer

```
void interruzione_timer(void) interrupt 1{
 //fermo
 TR0=0;
 TF0=0;
 loops--;
 if (loops<=0){
         //devo cambiare lo stato del led
     if(led_status){
       //accendo
       P1_6=1;
       led_status=0;
       loops=loops_on;
     }
     else{
       //spengo
       P1_6=0;
       led_status=1;
       loops=n_loops-loops_on;
       //faccio ripartire
     }
 }
 TR0=1;
}
```

Dunque si è definita la luminosità come un int che può essere 0, 25,50,75,100. La variabile lumstatus identifica in che stato si è tra i 5 possibili. Essa viene modificata nell'interrupt del bottone, quindi ogni volta che viene schiacciato.

Listing 1.7: Interrupt pulsante

```
void interruzione_pulsante(void) interrupt 19{
 TR0=0;
 TF0=0;
 //cambio status
 if(lum_status<4){
   lum_status++;
 }
 else{
```

```
    lum_status=0;
  }
  luminosity=lum_status*25;
  loops_on=n_loops*luminosity/100;
  //resetto tutto
  loops=loops_on;
  led_status=0;
  P1_6=1;
  //reset del flag dell'interrupt esterna 7
  P3IF &= ~0x80;
  TR0=1;//faccio ripartire il timer
}
```

Infine è necessario settare le variabili globali e quelle nel main

Listing 1.8: Variabili globali

```
#define n_loops 5
char idata stack1[16];
int luminosity = 0;
int loops_on;
int loops; //lo stato iniziale=1
int led_status=0;
int lum_status=0;//indica le opzioni(0,25, 50, 75,100)
```

Listing 1.9: main

```
SP = (char) (&stack1);
Init_Device();
loops_on=n_loops*luminosity/100;
loops=loops_on;
```

## 1.3 Relazione Seriale

La comunicazione UART è l'oggetto di questa esperienza. Si è messo in comunicazione il micro con il computer, tramite un cavo che collega la usb del computer con i due pin(più il pin GND) del micro, uno per la trasmissione e uno per la ricezione.

Prima di trasmettere e ricevere un carattere è necessario innanzitutto configurare la UART, con il BAUDE RATE(9600 bps) corretto, e con il timer corretto e infine abilitare l'interrupt.

Per trasmettere è necessario mettere in SBUF0 il carattere che si vuole mandare, il micro provvederà a trasmetterlo e a trasmissione conclusa genererà un'interrupt settando la flag TI(trasmissione completata). Per quanto riguarda la ricezione invece, il programma entra in interrupt con la flag RI(ricezione completata) dopo aver ricevuto il bit di stop, e mette a disposizione il carattere ricevuto nello SBUF0.

### 1.3.1 Seriale in C

In una prima fase del programma il micro trasmetteva un messaggio di benvenuto, nella seconda fase entrava in modalità eco, quindi ad ogni carattere che gli si mandava lui trasmetteva lo stesso carattere. Infine nell'ultima fase memorizzava una serie di caratteri(massimo 10) e dopo un certo carattere("#") ritrasmetteva la stessa serie. Per realizzare ciò è stata implementata la funzione scelta che a seconda del valore di status(0,1 o 2) svolge compiti diversi. Nella prima trasmette il messaggio a cui punta *puntatore*(nella prima fase è il messaggio di benvenuto ma successivamente diventa il messaggio trasmesso dall'utente al computer), nella seconda viene implementata la modalità eco e nella terza si riceve una serie di caratteri che viene memorizzata in un vettore. Se il messaggio mandato dall'utente è più lungo di 10 caratteri viene trasmesso un messaggio di errore.

Listing 1.10: Funzione scelta

```
void scelta(void){
  //controllo lo status
  switch(status){
    case(0):
      //trasmetto il messaggio
      i++;
      if (i<lenght_da_trasmettere){
        SBUF0=*(puntatore+i);
      }
      else{
        //ho finito di trasmettere entro in ricezione
        status++;
        i=0;
        REN0=1;
      }
      break;
    case(1):
      key=SBUF0;//ricevo, leggo SBUF0
      RI0=0;
      if(key=='#'){
        i=2;
        status++;
      }
      else{
        if(!loaded){
          loaded=1;
          SBUF0=key;
        }
      }
```

```
      break;
    case(2):
      RI0=0;
      if(i<ML+3){
        key=SBUF0;
        //aggiungo il carattere letto al vettore
        msg_ricevuto[i]=key;
        i++;
        if(key=='#'){
          REN0=0; //smetto di ricevere
          status=0;
          puntatore=msg_ricevuto;
          msg_ricevuto[i-1]=CR;
          msg_ricevuto[i]=LF;
          lenght_da_trasmettere=i+1;
          i=0;
          SBUF0=*puntatore;//trasmetto messaggio
        }
      }
      else{
        REN0=0;
        i=0;
        status=0;
        puntatore=msg_errore; //trasmetto errore
        lenght_da_trasmettere=lenght_errore;
        SBUF0=*puntatore;
      }
      break;
  }
}
```

La funzione scelta viene chiamata solo dall'interrupt della UART, a prescindere che si sia finito di trasmettere o di ricevere.

Listing 1.11: Interrupt UART

```
void UART0() interrupt 4{
  if(TI0==1){
    //ho appena finito di trasmettere
    scelta();
    //dice che ha caricato l'ultimo carattere
    loaded=0;
    //resettare flag
    TI0=0;
    return;
  }
  else if(RI0==1){
```

```
    //ho appena finito di ricevere
    scelta();
    return;
  }
  else{
    //ho un problema
    return;
  }
}
```

Listing 1.12: Variabili globali

```
#define CR 13
#define LF 10
#define ML 10
uchar key;
uchar idata msg_trsm[]={'C','i','a','o',CR,LF};
uchar lenght_da_trasmettere = 6;
uchar idata msg_errore[]={CR,LF,'E','r',
'r','o','r','e',CR,LF};
uchar lenght_errore=10;
uchar i=0;
uchar * puntatore;
uchar idata msg_ricevuto[ML+4];
uchar status=0;
uchar loaded=0;
```

Il tutto inizia trasmettendo il primo carattere del messaggio di benvenuto, una volta che la trasmissione è stata completata si passa al carattere successivo e così via fino alla fine. Al termine di queste istruzioni il programma entra in un ciclo infinito ed esce solo quando l'utente trasmette dei caratteri, entrando nell'interrupt e quindi nella funzione scelta.

Listing 1.13: main

```
SP=(char)&stack;
Init_Device();
//inizia trasmissione msg di benvenuto
puntatore = msg_trsm;
SBUF0=*puntatore;
msg_ricevuto[0]=CR;
msg_ricevuto[1]=LF;
while(1);
```

### 1.3.2 Seriale in assembly

Questo semplice programma, è una piccola variante di quello precedente in cui si vuole imparare a gestire un vettore in Assembly. Infatti ciò che si definisce è un vettore nella memoria programmi, questo perché il vettore non verrà modificato ma solo letto, contenente il messaggio di benvenuto. L'indirizzo del vettore viene poi passato, subito dopo la configurazione del micro, al Data Pointer. Viene successivamente implementata una funione NEXT che salva nell'Accumulator il carattere successivo in modo tale che sia accessibile dal main del programma in c. Dunque nel programma in C basterà chiamare questa funzione e prendere il carattere contenuto in A e trasmetterlo.

Listing 1.14: Vettore in Assembly

```
VETTORE SEGMENT CODE
STACK SEGMENT IDATA
NEXT SEGMENT CODE
RSEG STACK
STACK2: DS 10H
RSEG VETTORE
VETT: DB 'Ciao',00H
RSEG NEXT
N:  MOVC A,@A+DPTR
    RET
RSEG CODICE
START:
  LCALL Init_Device
  MOV DPTR, #VETT
  RET
END
```

Listing 1.15: codice in C

```
NEXT();
*puntatore =(uchar)ACC;
```

# Capitolo 2

# ARM

## 2.1 Introduzione ai micro ARM e CORTEX

### 2.1.1 Configurazione del micro

## 2.2 Seriale

L'esperienza della seriale è del tutto analoga a quella con l'8051, il programma è diviso in 3 fasi:

1. Nella prima fase si trasmette un messaggio di benvenuto al computer
2. Nella seconda fase il micro entra in modalità eco, ritrasmettendo indietro tutti i caratteri che gli si mandano
3. Infine riceve una serie di caratteri memorizzandoli e poi rimandando il messaggio ricevuto

Per la USART è stata usata la USART2 del micro abilitandola dal STM32Cube e se ne è abilitato l'interrupt nel NVIC(Nested Vector Interrupt Controller), questo perché la USART2 è già implementata nell'STLink quindi la comunicazione avviene tramite il cavo mini-USB.

Un'ulteriore differenza con l'8051 è che il buffer per la trasmissione e per la ricezione sono distinti(il che probabilmente rende più chiaro anche il codice) e sono rispettivamente *TDR* e *RDR*.

Si è creato un file_condiviso.h come header sia per il file main.c sia per il file che gestisce gli interrupt(stm32f0xx_it.c). Esso contiene alcune definizioni importanti per entrambi i codici e permette il dialogo tra i due file, come ad esempio *puntatore* che viene usata da entrambi.

Listing 2.1: Header dei files

```
#define CR 13
#define LF 10
```

```
#define ML 10
extern unsigned char * puntatore;
extern unsigned char msg_ricevuto[ML+4];
extern int lenght_da_trasmettere;
extern unsigned char  messaggio_benvenuto[7];
extern int lenght;
extern unsigned char msg_errore[10];
extern int lenght_errore;
extern int status;
extern int i;
extern int loaded;
```

Quelle riportate di seguito sono le inizializzazioni delle variabili precedenti nel main.

Listing 2.2: Inizializzazione variabili

```
unsigned char * puntatore;
unsigned char msg_ricevuto[ML+4];
int lenght_da_trasmettere=6;
unsigned char messaggio_benvenuto[]={'C','i','a'
,'o',CR,LF};
unsigned char;
msg_errore[]={CR,LF,'E','r','r','o','r','e',CR,LF};
int lenght_errore=10;
int status = 0;
int loaded =0;
int i=0;
```

Il vero codice inizia dopo che è avvenuta la configurazione del micro, abilitando l'interrupt di trasmissione e ricezione della USART2. Successivamente si passa il primo carattere da trasmettere al registro TDR.[1]

Listing 2.3: main

```
//reset del interrupt control register
//al bit trasmission completed
USART2->ICR |= USART_ICR_TCCF;
//abilito l'interrupt per la trasmissione
USART2->CR1 |= USART_CR1_TCIE;
//abilito l'interrupt per la ricezione
USART2->CR1 |= USART_CR1_RXNEIE;
//inizio a trasmettere il messaggio di benvenuto
msg_ricevuto[0]=CR;
msg_ricevuto[1]=LF;
puntatore= messaggio_benvenuto;
```

[1] È necessario fare il reset del bit di transmission completed perché quando vengono abilitati gli interrupt il micro entra in interrupt e se questo bit fosse 1 lo interpreterebbe come primo carattere trasmesso

```
USART2->TDR=*puntatore;
```

Quando il micro finisce di trasmettere entra nell'interrupt con il bit di TC ad 1, a questo punto è necessario fare qualcosa e sarà la funzione scelta a determinare cosa.

Listing 2.4: gestione dell'interrupt

```
if((USART2->ISR & USART_ISR_TC) == USART_ISR_TC){
  //resetto il flag di trasmissione completata
  USART2->ICR |= USART_ICR_TCCF;
  //allora ho appena finito di trasmettere
  scelta();
  loaded=0;
  return;
  }
else if((USART2->ISR&USART_ISR_RXNE)==USART_ISR_RXNE){
  //allora ho appena finito di ricevere
  scelta();
  return;
  }
```

La funzione scelta è il cuore di questo programma. È l'evoluzione della stessa implementata per la seriale in C 1.10.

Listing 2.5: Funzione scelta

```
void scelta(){
  unsigned char carattere;
  switch(status){
    case 0:
      //inserisco nel buffer di trasm il carattere
      i++;
      if(i<lenght_da_trasmettere){
        USART2->TDR=*(puntatore+i);
      }
      else{
        status=1;
        i=0;
      }
      break;
    case 1:
      //do
        //inserisco nel buffer di trasmissione
        //il carattere trovato nel buffer in ricezione
        carattere = (char)USART2->RDR;
        if(carattere=='#'){
          i=2;
          status=2;
```

```
        }
        else{
        if(!loaded){
          loaded=1;
          USART2->TDR=carattere;
        }
      }
      break;
    case 2:
      //do

        carattere=USART2->RDR;
        if(i<ML+3){
          //aggiungo il carattere letto al vettore
          msg_ricevuto[i]=carattere;
          i++;
          if(carattere=='#'){
            //smetto di ricevere
            status=0;
            puntatore=msg_ricevuto;
            msg_ricevuto[i-1]=CR;
            msg_ricevuto[i]=LF;
            lenght_da_trasmettere=i+1;
            i=0;
            //trasmetto il messaggio ricevuto
            USART2->TDR=*puntatore;
          }
        }
        else{
          status=0;
          //trasmetto il messaggio di errore
          puntatore=msg_errore;
          lenght_da_trasmettere=lenght_errore;
          i=0;
          USART2->TDR=*puntatore;
        }

      break;
  }
}
```

## 2.3 Timer7 e Led

Questa esperienza è analoga a quella svolta con il led ed il bottone per l'8051 in C(referenza:1.2.3). Infatti l'obiettivo è di realizzare un programma in cui si possa

cambiare la luminosità del LED sfruttando gli interrupt del timer del micro ed il bottone. Dunque è stato configurato il Timer 7(TIM7) a 16 bit e l'overflow a 10000(quindi il timer conta da 0 a 10000), con il relativo interrupt. Abbiamo abilitato la porta PA5 come GPIO output per poter accendere e spegnere il led, ed è stato abilitato l'interrupt esterno della porta PC13(il bottone), di modo che ogni volta che viene premuto, viene generato un interrupt.

La funzione wait_tim aspetta che il timer abbia raggiunto n interrupt, facendo trascorrere un po' di tempo.

Listing 2.6: Funzione wait_tim per aspettare

```
void wait_tim(){
  wait=1;
  TIM7->CNT=0;//si imposta il contatore a 0
  TIM7->SR=0;//si imposta lo status register a 0
  //si abilita il counter per iniziare a contare
  TIM7->CR1|=TIM_CR1_CEN;
  //si aspetta fino a che non viene settata a 0
  while(wait){}
}
```

All'interno dell'interrupt del timer è necessario fare un check sulla variabile *loop*, la quale indica quanti cicli di interrupt bisogna attendere prima di fermare il timer.

Listing 2.7: Gestione dell'interrupt del timer

```
if(loops>0){
    loops-=1;
    TIM7->CR1|=TIM_CR1_CEN;
  }
else{
  wait=0;//si puo smettere di aspettare
  //e smettere di contare
  TIM7->CR1&=~TIM_CR1_CEN;
}
```

All'interno dell'interrupt del bottone è necessario cambiare la luminosità, questo come nel caso della luminosità in C1.7, viene fatto cambiando il numero di loop che il led sta accesso ovvero la variabile *loop_on*. Come nel caso del C, la variabile status indica 5 possibili luminosità(0,25,50,75,100), ciclando su di essi.

Listing 2.8: Gestione interrupt del bottone

```
if(status<4){
  status++;
}
else{
  status=0;
```

```
}
loops_on=n_loops*status*25/100;//cambio la luminosita'
//resetto
wait=0;
acceso=0;
```

Listing 2.9: Variabili globali interrupt

```
int status=0;
int loops_on=0;
int n_loops=50;
int acceso;
int wait;
volatile int loops=0;
```

Listing 2.10: Variabili globali main

```
extern int loops_on;
extern int n_loops;
extern int acceso;
extern int wait;
extern int loops;
```

All'interno del main è necessario innanzitutto abilitare l'interrupt del timer, successivamente si entra in un loop infinito all'interno del quale si continua ad accendere e spegnere il led. Il led rimane acceso per *loops_on* cicli e spento per *n_loops-loops_on* cicli, con *n_loops* il numero massimo di cicli.

Listing 2.11: Main

```
//abilito l'interrupt del timer
TIM7->DIER|=TIM_DIER_UIE;
while (1)
  {
  if(acceso==0){
    //accendo
    //cambio lo stato del led
    HAL_GPIO_TogglePin(GPIOA,GPIO_PIN_5);
    //devo aspettare loops_on cicli
    loops=loops_on;
    wait_tim();
    acceso=1;
  }
  else{
    //spengo
```

```
    HAL_GPIO_TogglePin(GPIOA,GPIO_PIN_5);
    //devo aspettare loops massimi - loops_on
    loops=n_loops-loops_on;
    wait_tim();
    acceso=0;
  }
}
```

## 2.4 I2C, termometro e display

### 2.4.1 Configurazione del micro

L'esperienza riguardante il protocollo di trasmissione I2C vede coinvolta, oltre che al micro STM32F072rb, anche una scheda da posizionare sopra la scheda di sviluppo Nucleo, al di sopra della quale è connesso un LCD, un termometro ed una manopola, insieme a 4 led. Obbiettivo dell'esperienza è mettere in comunicazione il micro con il display ed il termometro attraverso l'i2c e leggere il potenziale registrato dalla manopola attraverso l'adc. Prima di iniziare a scrivere il codice è stato necessario configurare attraverso CUBE il micro, essendo la scheda posizionata sui pin del nucleo bisogna selezionare i pin adibiti alla comunicazione I2C, quelli per l'ADC e i pin dei led. In particolare le operazioni eseguite sono le seguenti:

- Attivare l'I2C1 sui pin PB8 e PB9
- Abilitare PA5 come GPIO_Output, per il reset del display
- Abilitare PA9 come GPIO_Output, per la retroilluminazione del display
- Abilitare PC0 e PC3 come GPIO_Output, servono per il termometro. (Notare che i pin di comunicazione I2C sono gli stessi sia per il display che per il termometro, è una caratteristica intrinseca del protocollo I2C avere solo due linee adibite alla comunicazione multiutente, una per il clock e una per i dati)
- Abilitare PA0 come ADC_IN0 ovvero come porta per l'ADC
- Le porte PB0,PB1,PB2,PB3 sono messe come GPIO_Output per comandare i 4 led

È stato ovviamente abilitato l'interrupt dell'ADC, dell'I2C, del Timer7(con un autoreload value di 48000, ovvero parte da 0 e arriva a 48000) in modo che con il clock a 48MHz andasse in overflow ogni 1ms. La frequenza dell'ADC è stata invece impostata a 14MHz. Per una seconda parte dell'esperienza è stata anche attivata la USART2 come nelle precedenti esperienze(freq. a 9600bit/s...) con il relativo interrupt.

### 2.4.2 Esperienza

Descrizione del programma I2C display termometro. L'obiettivo di questo programma è scrivere su un display la temperatura registrata dal termometro e il potenziale registrato da un potenziometro a manopola letto dall'adc. è necessario dividere il programma in più parti:

- Leggere dall'ADC il potenziale
- Inizializzare il display
- Leggere la temperatura dal termometro
- Trasmettere al display la temperatura
- Trasmettere al display il potenziale

### 2.4.3 I2C

Essendo l'I2C triggerato sia quando si sta inizializzando il display, sia quando si sta leggendo la temperatura sia quando si deve trasmettere la temperatura letta, si è ritenuto necessario l'ausilio di , simile a quella già implementata in 1.10 e in 2.5. L'interrupt viene triggerato nei seguenti casi:

1. Si è specificato di voler trasmettere NBYTES di caratteri e la trasmissione si è conclusa(STOPIE)
2. Si è inviato un singolo carattere e la trasmissione è andata a buon fine(TXIE)
3. Si è specificato di voler leggere un dato che è stato ricevuto(RXIE)

Per inizializzare correttamente il display è necessario mandargli una serie di comandi ed impostazioni intervellate da 1 ms. Per questa ragione è necessario che la comunicazione avvenga per blocchi di 2 bytes, 1 byte per dire al display che quello successo è un comando ed 1 byte per il comando. Tra i blocchi da 2 bytes si attendono per sicurezza 100ms, poi si pulisce la stop flag e si fa ripartire la comunicazione. Per rendere più facilmente interpretabile lo stato del micro, alla conclusione di ogni sezione di codice vengono accesi i led.

Listing 2.12: Gestione Interrupt I2C per inizializzazione Display

```
if((I2C1->ISR & I2C_ISR_STOPF)!=0){
    //ho finito di trasmettere la serie di
    //caratteri/comandi
  i2c_loops=100;
  while(i2c_loops>0){
    TIM7->SR=0;
    TIM7->CNT=0;
    TIM7->CR1|=TIM_CR1_CEN;
    while(TIM7->SR==0){}
```

```
    i2c_loops--;
  }
  TIM7->CR1&=~TIM_CR1_CEN;
  if(j<16){
      //sto trasmettendo ancora i bytes
      //di inizializzazione
      //Pulisco la flag dello stop-bit
    I2C1->ICR |= I2C_ICR_STOPCF;
    //Faccio partire lo START
    I2C1->CR2 |= I2C_CR2_START;
  }
}
if (j<16){
  if(j==0){
    led();
  }
  //viene mandato un carattere
  I2C1->TXDR = initdata[j];
  j++;
}
```

Per far partire l'inizializzazione è necessario prima abilitare gli interrupt dell'i2c, poi specificare quanti byte si vogliono trasmettere e a che indirizzo. Gli interrupt vengono impostati da CR1 mentre CR2 si occupa degli indirizzi delle periferiche e dei bytes da trasmettere o leggere. CR2 ha i bit che si occupano dei bytes da trasmettere dalla posizione 16 a 23 è necessario quindi ruotare a sx NBYTES di 16.

Listing 2.13: Inizio inizializzazione

```
/* definisco gli indirizzi*/
//indirizzo lcd, ruotato a sx per lasciare bit di r/w
#define LCD_ADDRESS (0X3E<<1)
//indirizzo termometro
#define LM76_ADDRESS (0X48<<1)

/* definisico le variabili di interesse globale*/

char initdata[]={0x00,0x38,0x00,0x39,0x00,0x14,
0x00,0x74,0x00,0x54,0x00,0x6F,0x00,0x0F,0x00,0x01};
int NBYTES =(2<<16);

/* faccio partire la trasmissione*/

I2C1->CR1|=I2C_CR1_RXIE;
I2C1->CR1|=I2C_CR1_TXIE;
I2C1->CR2|=I2C_CR2_AUTOEND;
```

```
I2C1->CR1|=I2C_CR1_STOPIE;

I2C1->CR2 |= NBYTES;
I2C1->CR2 |= LCD_ADDRESS;
I2C1->CR2 |= I2C_CR2_START;
```

Una volta inizializzato il display si può procedere con la trasmissione dei messaggi che si vuole visualizzare. Sono due i messaggi, uno indicante la temperatura ed uno indicante il potenziale della manopola(in una seconda versione verrà riportata la percentuale invece della tensione). Il primo messaggio da mandare è quello relativo alla temperatura per semplicità del codice, infatti dopo che sono stati trasmessi i due messaggi ogni 10 secondi verrà verificata la posizione della manopola aggiornando il valore della tensione. Occorre prima leggere la temperatura, impostando l'i2c in lettura di 2 bytes. Ciò viene fatto dentro all'interrupt dell'i2c.

Listing 2.14: Lettura temperatura

```
//definisco il messaggio della temperatura
char temp[]={'T','e','m','p',':','.','.','.','.'};

switch(j){
case 16:
     I2C1->ICR |= I2C_ICR_STOPCF;
     I2C1->CR2 &= ~0xFF;
     I2C1->CR2 |= LM76_ADDRESS;
     NBYTES = 2 << 16;
     I2C1->CR2 &= ~(0xFF<<16);
     I2C1->CR2 |= NBYTES;
     //setto il bit di lettura
     I2C1->CR2 |= I2C_CR2_RD_WRN;
     I2C1->CR2 |= I2C_CR2_START;
     j++;
     break;
case 17:
     if(I2C1->ISR & I2C_ISR_RXNE){
     t1 = I2C1->RXDR; //prima parte della temperatura
     j++;
     }
     break;
 case 18:
     if(I2C1->ISR & I2C_ISR_RXNE){
     t2= I2C1->RXDR; //seconda parte della temperatura
     j++;
     }
     break;
```

```
 case 19:
     t1= t1<<8;
     t= ((t1+t2)>>3);
     temp[5]=(int)t/160+'0';
     resto=t%160;
     temp[6]=(int)(resto*10)/160+'0';
     resto=(resto*10)%160;
     temp[8]=(int)(resto*10)/160+'0';
     puntatore=temp;
     lenght_da_trasmettere=9;
     j++;
     break;
```

Ora che la temperatura è stata recuperata si mette il micro in comunicazione con il display, in trasmissione di 12 bytes(3 di impostazione e 9 caratteri del messaggio). I comandi trasmessi sono 0x80 indicante che i successivi 2 bytes sono di controllo, 0x80 indicante l'inizio della prima riga e 0x40 per specificare il tipo di dati che verranno mandati(caratteri). Ogni volta che l'interrupt viene triggerato j viene incrementato e viene trasmesso un nuovo carattere.

Listing 2.15: Trasmissione messaggio display

```
case 20:
  I2C1->ICR |= I2C_ICR_STOPCF;
  //si mette il trasmissione
  I2C1->CR2 &= ~I2C_CR2_RD_WRN;
  I2C1->CR2 &= ~0xFF;
  I2C1->CR2 |= LCD_ADDRESS;
  NBYTES = ((lenght_da_trasmettere+3) <<16);
  I2C1->CR2 &= ~(0xFF<<16);
  I2C1->CR2 |= NBYTES;
  I2C1->CR2 |= I2C_CR2_START;
  j++;
  break;
case 21:
  I2C1->TXDR = 0x80;
  j++;
  break;
case 22:
   I2C1->TXDR = 0x80;
   j++;
   break;
case 23:
   I2C1->TXDR = 0x40;
   j++;
   break;
case 24:
```

```
   if(i<lenght_da_trasmettere){
    I2C1->TXDR = *(puntatore+i);
    i++;
   }
   else{
    j++;
        led();
  }
  break;
```

Adesso bisogna trasmettere il messaggio della manopola. In questa versione viene usata la percentuale del potenziale(rispetto a 3.3) a cui è ruotata(la versione precedente mandava il vettore vec che contiene il potenziale letto dall'adc). Al termine dell'invio e dopo che sono passati 10 secondi viene controllato nuovamente il potenziale della manopola e ne viene aggiornato il valore.

Listing 2.16: Trasmissione messaggio manopola

```
case 25:
  led();
  i2c_loops=100;
  while(i2c_loops>0){
    TIM7->SR=0;
    TIM7->CNT=0;
    TIM7->CR1|=TIM_CR1_CEN;
    while(TIM7->SR==0){}
    i2c_loops--;
  }
  TIM7->CR1&=~TIM_CR1_CEN;
  //converto in percentuale vec
  pot=(float)vec[0]+(float)vec[1]/10
  +(float)vec[2]/100;
  perc[0]=(int)(pot*10)/33;
  resto=((int)(pot*10))%33;
  perc[1]=(int)(resto*10)/33;
  resto=(resto*10)%33;
  perc[2]=(int)(resto*10)/33;
  resto=(resto*10)%33;
  perc[3]=(int)(resto*10)/33;
  //controllo se ci sono delle cifre iniziali nulle
  //nel qual caso trasmetto uno spazio
  if(perc[0]==0){
    msg[4]='␣';
    if(perc[1]==0){
      msg[5]='␣';
    }
    else{
```

```
      msg[5]=perc[1]+'0';
    }
  }
  else{
    msg[4]=perc[0]+'0';
    msg[5]=perc[1]+'0';
  }
  msg[6]=perc[2]+'0';
  msg[8]=perc[3]+'0';
  lenght_da_trasmettere=10;
  puntatore= msg;
  i=0;
  I2C1->ICR |= I2C_ICR_STOPCF;
  NBYTES = ((lenght_da_trasmettere+3) <<16);
  I2C1->CR2 &= ~(0xFF<<16);
  I2C1->CR2 |= NBYTES;
  I2C1->CR2 |= I2C_CR2_START;
  j++;
  break;

case 26:
  I2C1->TXDR = 0x80;
  j++;
  break;

case 27:
    //inizio della seconda riga del display
    I2C1->TXDR = 0xC0;
    j++;
    break;

case 28:
    I2C1->TXDR = 0x40;
    j++;
    break;
case 29:
    if(i<lenght_da_trasmettere){
        I2C1->TXDR = *(puntatore+i);
      i++;
     }
     else{
      //cambio puntatore
        //cambio lenght_da_trasmettere
      led();
      i2c_loops=10000;
        while(i2c_loops>0){
```

```
          TIM7->SR=0;
          TIM7->CNT=0;
          TIM7->CR1|=TIM_CR1_CEN;
          while(TIM7->SR==0){}
          i2c_loops--;
        }
        TIM7->CR1&=~TIM_CR1_CEN;
        ADC1->CR|=ADC_CR_ADSTART;
        j=25;
    }
     break;
```

### 2.4.4 ADC

Per quanto riguarda l'ADC è stato necessario prima configurarlo ovvero abilitare gli interrupt, eseguire una procedura(leggermente lunga) di calibrazione, selezionare il canale 17 da cui leggere la tensione VREFINT_DATA, ovvero un valore che permetterà di ottenere un valore più accurato per la tensione della manopola. Una volta letto il dato VREFINT_DATA si aspetta 1 ms e si legge il dato della manopola, riscalandolo con la seguente formula:

$$V_{channel} = \frac{3.3 \times VREFINT\_CAL \times V_{channel0}}{VREFINT\_DATA \times 4095} \tag{2.1}$$

Dove VREFINT_CAL è un valore memorizzato alla cella di memoria 0x1FFFF7BA al momento della fabbricazione. Una volta calcolato il valore più preciso della tensione della manopola, se ne prendono le 3 cifre più significative e le si memorizzano in vec per essere trasmesse al display 2.16.

Listing 2.17: Gestione interrupt ADC

```
valore=*(uint16_t*)0x1FFFF7BA;
//controllo se ha finito di convertire
if((ADC1->ISR & ADC_ISR_ADRDY)!=0){
  //resetto l'arddy
  ADC1->ISR|=ADC_ISR_ADRDY;
  //faccio partire la conversione
  ADC1->CR|=ADC_CR_ADSTART;
}
if((ADC1->ISR & ADC_ISR_EOC) == ADC_ISR_EOC){
  if((ADC1->CHSELR & ADC_CHSELR_CHSEL0)!=0){
    //leggo dal registro ADC DR
    m=ADC1->DR;
    //ricalcolo il valore per 1V
    //m_1 dovrebbe essere circa 1241.2*10
    m_1=(int)10*(4095*vrefint_data)/(3.3*(valore));
    vec[0]=(int)(m*10/m_1); //m prima cifra
```

```
    resto=(m*10)%m_1;//resto
    vec[1]=(int)(resto*10/m_1);//m seconda cifra
    resto=(resto*10)%m_1;//resto
    vec[2]=(int)(resto*10/m_1);//m terza cifra
  }
  else{
    //leggo il dato per la calibrazione
    vrefint_data=ADC1->DR;
  }
}
```

Listing 2.18: Inizializzazione ADC

```
ADC1->IER|=ADC_IER_ADRDYIE;
ADC1->IER|=ADC_IER_EOCIE;

//abilito e configuro l'adc
//se fosse gia abilitato lo disabilito
//per fare la calibrazione
if ((ADC1->CR & ADC_CR_ADEN) != 0)
{
  ADC1->CR |= ADC_CR_ADDIS;
}
//aspetto che non sia piu enable
while ((ADC1->CR & ADC_CR_ADEN) != 0){}
//faccio la calibrazione
ADC1->CR|=ADC_CR_ADCAL;
//ed aspetto che abbia finito
while((ADC1->CR & ADC_CR_ADCAL)==ADC_CR_ADCAL){}
//se adrdy fosse 1 la metto a 0
if ((ADC1->ISR & ADC_ISR_ADRDY) != 0)
{
  ADC1->ISR |= ADC_ISR_ADRDY;
}
//abilito l'adc e spengo il canale 0
//quindi leggo solo dal 17
ADC1->CHSELR&=~ADC_CHSELR_CHSEL0;
ADC1->CR|=ADC_CR_ADEN;
//verra triggerato l'interrupt quando sara pronto

//adesso dovrebbe essere stato registrato
//il valore vrefint_data
//aspetto 1 millisecondo
loops=1;
wait_tim();
```

```
//cambio il canale da cui leggere il dato nell'adc
ADC1->CHSELR&=~ADC_CHSELR_CHSEL17;
ADC1->CHSELR|=ADC_CHSELR_CHSEL0;
//faccio ripartire la lettura
ADC1->CR|=ADC_CR_ADSTART;
```

## 2.5 Orologio, progetto personale

Listing 2.19: Comunicazione seriale ARM