SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA DELL' INFORMAZIONE
INFORMATICA E STATISTICA
Corso di Laurea Magistrale in Ingegneria Elettronica
Master Degree in Electronic Engineering

Sistemi Operativi

Relazione progetto: Oscilloscopio con Arduino

Student Name:

Andrea Marcelli 1901726

Data consegna relazione: 28/02/24

ACCADEMIC YEAR 2023/2024

Contents

1 Introduction

Per questo progetto ho usato una scheda contenente un Atmega2560 per creare un oscilloscopio secondo le specifiche fornite.
Le specifiche erano le seguenti:

- Possibility of configuring (from PC) which channel(s) are being sampled and the sampling frequency
- Possibility of operating in two modes
  - continuous sampling: (each sample is sent to the PC as it becomes ready)
  - buffered mode: a set of samples are stored locally and then sent in bulk
- Implementation of "trigger" (a condition that decides when to start sampling, in buffered mode)
- Serial communication should be asynchronous (use an interrupt on the UART)

e sono rispettate nei seguenti modi:

- Dal programma PC verrà chiesto all'utente di selezionare, durante l'esecuzione, il numero di canali che si vogliono utilizzare (da 1 a 8) e successivamente ogni quanti decimi di ms effettuare il sampling. Quindi invece di chiedere direttamente la frequenza ho scelto di chiedere il suo inverso all'utente.
- Dal programma PC verrà chiesto all'utente di selezione in che modalità di operazione vuole operare, avendo la possibilità di scegliere tra continous mode o buffered mode.
- Dal programma PC verrà chiesto all'utente quando iniziare le conversioni, per entrambe le modalità di operazione
- Per gestire la comunicazione dal lato Atmega sono stati utilizzati i relativi interrupt di ricezione e trasmissione.

Il progetto consiste dei seguenti file:

- ./AVR/ cartella che contiene il progetto per l'atmega
  - main_avr.c contiene il main del progetto per l'atmega
  - Makefile
  - avr_common/
    - my_adc.c contiene le funzioni relative all'ADC e al timer
    - my_adc.h
    - my_uart.c contiene le funzioni relative alla uart
    - my_uart.h
    - my_variables.h contiene le variabili esterne
    - avr.mk
- ./PC/
  - main_pc.c contiene il main del progetto per PC
  - serial_linux.c contiene le funzioni relative alla comunicazione seriale
  - serial_linux.h
  - Makefile
  - oscilloscopio.plt contiene lo script per GNUPLOT

2 Implementazione e Commento del codice

2.1 Atmega

Il programma dell'Atmega ho deciso di strutturarlo come una macchina a stati, governata dalla variabile "state" che contiene lo stato presente. Il main quindi conterrà solo l'inizializzazione delle varie periferiche, l'abilitazione degli interrupt e infine un loop infinito che chiama la macchina a stati.

Prima di descrivere l'utilizzo delle periferiche e la macchina a stati, è importante definire le caratteristiche delle due modalità.
Queste sono limitate da due elementi differenti:

- Buffered mode: visto che i campioni devono essere salvati prima di essere inviati tutti insieme alla fine delle conversioni, l'elemento limitante è sicuramente la SRAM del microcontrollore, di soli 8KB. Nonostante possa operare indisturbato da eventuali trasferimenti UART, al contrario di come sarà per la modalità continous, il periodo di campionamento non può essere molto basso, perchè una frequenza alta va a riempire molto velocemente il buffer disponibile. Si è quindi definito sul programma PC, tramite una macro, una lunghezza massima di 7750, per essere sicuri di lasciare abbastanza spazio alle altre variabili, e da questa, dal tempo totale di operazione, e dai valori passati dall'utente, si calcola il periodo minimo accettabile.

- Continous mode: qui, al contrario, la memoria rimane libera visto che i dati trovati devono essere subito inviati al PC tramite UART. Il fattore limitante si è rivelato essere la velocità di trasferimento. Per mitigare il problema è stata inserita la possibilità di selezionare un baudrate di 38400 tramite l'apposita macro. Per informazioni su dove trovare le macro e come cambiarla si può fare riferimento alla sezione "Come compilare il progetto e Testarlo".

Visto i due limiti, spaziali per la buffered mode e temporali per la continous mode, si è scelto di implementare una modalità di esecuzione "approssimata" selezionabile tramite la macro APPROX. Nel datasheet, nella sezione "26.8.4 ADCL and ADCH – The ADC Data Register" si può leggere che "When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision (7 bit + sign bit for differential input channels) is required, it is sufficient to read ADCH.", con la seguente immagine a precedere l'affermazione:

26.8.4.1 ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

26.8.4.2 ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

In questa seconda modalità di esecuzione quindi, si andrà ad impostare l'ADC in modo che restituisca i risultati left adjusted e si andranno a leggere solo gli 8 bit più significativi, invece che i normali 10 bit che compongono il risultato completo.
Questo permette di migliorare notevolmente le prestazioni, andando a dimezzare il periodo minimo selezionabile per il sampling nel caso della continous mode, e andando a raddoppiare il numero di conversioni che il buffer interno può contenere nella buffered mode (dimezzando anche qui di conseguenza il periodo minimo selezionabile).

2.1.1 ADC

Per quanto riguarda l'ADC, questo è stato utilizzato senza il suo interrupt visto che le tempistiche di sampling sono state gestite da un timer, vedere il paragrafo successivo per maggiori informazioni su questo

aspetto. Tutte le funzioni che riguardano l'ADC sono presenti nel file my_adc.c.
L'inizializzazione avviene attraverso la seguente funzione:

```
void adc_init(void){
   //inizializzo ADC:

#ifdef APPROX
   ADMUX |= (1 << REFS0) | (1 << REFS1) | (1<<ADLAR); //left adjusted data, leggero' solo gli 8 bit piu' significativi del risultato (su 10 disponibili), contenuti in ADCH
#else
   ADMUX |= (1 << REFS0) | (1 << REFS1);          //da modificare in seguito per decidere quale/i ADC usare.
#endif
   ADCSRA |= (1 << ADEN);
}
```

Le uniche cose che vengono impostate sono il voltaggio di riferimento, selezionato tramite i bit "REFS0" e "REFS1", e in questo caso impostato alla sorgente interna di riferimento a 2.56V, la scrittura left adjusted dei risultati nel caso di modalità approssimata, e l'accensione dell'ADC tramite "ADEN".

Successivamente, dopo che l'utente seleziona il numero di canali da utilizzare viene chiamata la seguente funzione per disabilitare gli input buffer digitali dei pin scelti come ingressi analogici. Questo viene suggerito dal datasheet per risparmiare potenza.

```
void adc_sel(uint8_t adc_number){

   //disattivo gli input digitali (disattivo il buffer dell'input digitale)
   uint8_t i = 0;
   for(i=0;i<adc_number;i++){
      DIDR0 = (DIDR0<<1) | 0x1;
   }
}
```

Per effettuare le conversioni viene usata la funzione "void adc_conv(uint8_t var)", che prende come argomento il numero del canale da cui leggere l'ingresso analogico. La semplice conversione però ha prodotto problemi in fase di testing. Il problema si è rivelato essere la capacità interna al circuito di sampling dell'ADC, che non faceva in tempo a scaricarsi e a caricarsi per leggere il valore corretto in caso di valori molto diversi tra due pin consecutivi. Per risolvere il problema, ogni volta che si doveva effettuare una conversione, e quindi leggere l'ingresso applicato ad un canale ADC, si effettua la seguente routine:

```
adc_conv_ground();
adc_conv(var);
```

Dove la prima funzione va a leggere un pin, selezionabile dal MUX dell'ADC, fisso a massa per scaricare la capacità interna e la seconda leggerà il canale corretto per due volte, in quanto con una volta sola non riesce a raggiungere valori più alti di 1.2V, sempre per colpa della velocità di carica della capacità:

```
void adc_conv(uint8_t var){

   ADMUX = (ADMUX & 0xe0) | (var & 0x07); //seleziono il canale da cui leggere
   ADCSRA |= (1<<ADSC);                //faccio partire la conversione

   while(ADCSRA & (1<<ADSC));          //aspetta che finisca la conversione

   ADCSRA |= (1<<ADSC);                //faccio partire la seconda conversione
   while(ADCSRA & (1<<ADSC));          //aspetta che finisca la conversione

#ifdef APPROX
   buffer_tx[0] = ADCH;
#else
   buffer_tx[0] = ADCL;                 //va letto prima questo registro
   buffer_tx[1] = ADCH;
#endif
}

void adc_conv_ground(void){
   ADMUX = (ADMUX & 0xe0) | 0x1f;       //seleziono il canale di ground
   ADCSRA |= (1<<ADSC);                 //faccio partire la conversione
```

```
22
23    while(ADCSRA & (1<<ADSC));
24 }
```

Come si può vedere, i risultati della conversione sono salvati nell'array buffer_tx.
È importante sottolineare che queste conversioni extra non rallentano il programma, poichè la continous mode è limitata di molto dalla velocità della UART, mentre la buffered mode non raggiunge frequenze tali di sampling per la limitazione spaziale.

2.1.2 Timer

Il timer viene utilizzato nel programma per scandire il periodo di sampling. Usando il metodo visto in classe per inviare un interrupt ogni n millisecondi si è fatto inviare un interrupt ogni periodo di campionamento. L'interrupt è definito nel seguente modo:

```
1 ISR(TIMER5_COMPA_vect)
2 {
3     //Interrupt che si attiva quando il timer matcha l'OCR
4     ocr_int = 1;
5
6 }
```

Per farlo il più breve possibile, questo mette solo a 1 una variabile che andrà, nella macchina a stati, a far partire le conversioni.
Il timer è stato inizializzato nel seguente modo:

```
1 void timer_init(void){
2    //inizializzo timer: prescaler 256x, CTC
3    TCCR5A = 0;
4    TCCR5B = (1 << WGM52) | (1 << CS52);
5 }
```

con il prescaler impostato a 256 e l'opzione di CTC attiva per resettare il counter ogni compare match.
Infine, quando l'utente seleziona il periodo di sampling viene chiamata la seguente funzione per impostare il valore dell'output compare register:

```
1 void period_set(uint16_t period){
2    //imposto il periodo selezionato
3    uint16_t ocrval=(uint16_t)(6.25*period); //ogni quanti decimi di ms deve essere effettuato
         un sampling
4    OCR5A = ocrval;
5 }
```

Il periodo di campionamento viene passato in decimi di millisecondi per offrire più precisione di selezione sulle frequenze.

2.1.3 UART

Come da specifiche l'UART viene utilizzata tramite interrupt. Questa viene quindi inizializzata nel seguente modo:

```
1 void UART_init(void) {
2   // Set baud rate
3   UBRR0H = (uint8_t)(MYUBRR>>8);
4   UBRR0L = (uint8_t)MYUBRR;
5
6   UCSR0C = (1<<UCSZ01) | (1<<UCSZ00); /* 8-bit data */
7   UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0) | (1<<TXCIE0); /* Enable RX and TX */
8 }
```

Impostando il baudrate, selezionando il formato del byte inviato e abilitando transmitter e receiver con i relativi interrupt. I loro interrupt sono così definiti:

```
1 ISR(USART0_RX_vect)
2 {
3     //Interrupt che si attiva quando e' stato ricevuto un byte
4     buffer_rx[byte_rec++] = UDR0;
```

```
}

ISR(USART0_TX_vect)
{
    //Interrupt che si attiva quando e' finita la trasmissione
    byte_tra = 1;
}
```

Quando viene ricevuto un byte, l'interrupt viene eseguito e registra il byte nel buffer di ricezione, incrementando di uno la variabile che tiene conto dei byte ricevuti. Si è scelto questo approccio vista la conoscenza del numero e dell'ordine dei byte che verranno ricevuti durante l'esecuzione.
Per quanto riguarda invece l'interrupt di trasmissione, quando una trasmissione viene conclusa, pone a 1 la variabile byte_tra, che verrà controllata dalla funzione per l'invio dei dati:

```
void UART_putChar(uint8_t c){
  // wait for transmission completed
  while ( ! byte_tra);
  // Start transmission
  UDR0 = c;
  byte_tra = 0;
}

void UART_putString(uint8_t* buf){
  for(uint32_t num = 0;num<len;num++){
    UART_putChar(*buf);
    ++buf;
  }
}
```

2.1.4 Macchina a stati

la variabile "state" è cosi definita nel main:

```
uint8_t state;          //contiene lo stato in cui mi trovo:
                        //000 = inizializzazione
                        //001 = aspettando numero di canali
                        //010 = aspettando modalita' di esecuzione
                        //011 = aspettando period
                        //100 = aspettando trigger
                        //101 = esecuzione continous sampling
                        //110 = esecuzione buffered mode
                        //111 = fine programma
```

ci saranno quindi 8 stati in totale. La macchina a stati è implementata tramite uno switch sulla variabile "state".

Lo stato 0 contiene solo il passaggio allo stato 1, e avviene dopo le inizializzazioni.
Lo stato 1 controlla se è stato ricevuto un byte, e in caso affermativo assegna il byte letto (che si trova nel buffer di ricezione) alla variabile che contiene il numero di canali dell'ADC che si andranno ad utilizzare. Infine chiama la funzione adc_sel vista prima e setta la variabile di stato a 2.
Lo stato 2 è identitico allo stato 1, solo che questa volta nel buffer di input ci sarà la modalità di esecuzione scelta dall'utente.
Lo stato 3 inizia allo stesso modo, aspettando la ricezione del valore del tempo di campionamento, diviso su due byte, e lo ricostruisce. Successivamente chiama la funzione period_set vista sopra e calcola i valori delle variabili max_conv e len. Queste contengono rispettivamente il numero di conversioni che avverrà per ogni canale ADC (max_conv) e il numero totale di byte che si andranno a ricavare dalle conversioni (len), quest'ultimo verrà utilizzato, nel caso della buffered mode, per allocare correttamente il buffer. Il buffer viene allocato con la funzione malloc.
Lo stato 4 attende l'arrivo del segnale di trigger da parte dell'utente. Quando questo viene ricevuto mette la variabile "byte_tra" uguale a 1 per far partire, quando servirà, la prima trasmissione, attiva l'interrupt del timer e a seconda della modalità scelta va allo stato 5 o allo stato 6.
Lo stato 5 e lo stato 6 come prima cosa controllano se la condizione di fine conversioni è stata raggiunta (num_conv == max_conv). In caso positivo, per la continous mode (stato 5) semplicemente vengono disattivati gli interrupt e viene messo "state = 7", per la buffered mode la fine delle conversioni indica il dover inviare i dati salvati nel buffer. Questi vengono inviati tramite la funzione UART_putString e,

una volta finita la trasmissione, il buffer viene deallocato. Quando invece la condizione è falsa e quindi le conversioni non sono ancora finite, il programma aspetta un interrupt del timer e quando lo riceve fa partire le conversioni con il procedimento indicato nella sezione dedicata all'ADC. Nel caso di continous mode i dati vengono inviati appena finita la conversione, mentre per la buffered mode questi vengono solo salvati.
Infine lo stato 7 è vuoto.

2.2 PC

Il programma per PC è strutturato sequenzialmente. Inizia aprendo la seriale, configurandola e settandola in modalità "blocking". Questa modalità è stata utile in fase di lettura dei dati dalla scheda al PC. Successivamente viene aperto il file di output, utilizzando il nome passato dall'utente come primo argomento.
A questo punto inizia la parte del codice dedicata alle opzioni da passare alla scheda. Queste sono

(nell'ordine in cui vengono richieste):

- Numero di canali ADC da utilizzare per la conversione, da 1 a 8, salvato nella variabile adc_num.
- Modalità di esecuzione, a scelta tra continous mode e buffered mode, salvato nella variabile mode.
- Periodo di campionamento, misurato in decimi di ms ($10^{-4}s$), salvato nella variabile period.
- Segnale di trigger, per indicare alla scheda quando iniziare la conversione dei segnali, salvato nella variabile trigger.

Queste richieste all'utente e l'invio alla scheda usano tutte circa la stessa struttura, qui di seguito riportata nel caso del numero di canali:

```
1  while_var = 0;
2  while(! while_var){
3    printf("Quanti canali ADC utilizzare?\nSelezionare un numero da 1 a 8:\n");
4    scanf("%hhu", &adc_num);
5    while ( getchar() != '\n' );
6    if(adc_num >= 1 && adc_num <= 8){
7      printf("Hai selezionato %hhx canali\n\n", adc_num);
8      ssize_t res_adc = write(fd, &adc_num, 1);        //invia adc_num
9      if(res_adc != 1){
10       printf("ERRORE SU INVIO NUMERO CANALI ADC\n");
11       return -1;
12     }
13     while_var = 1;
14   }
15 }
```

Il ciclo while esterno serve ad aspettare un input valido da parte dell'utente, quindi in questo caso un numero compreso tra 1 e 8. Il ciclo while interno, contenente il getchar() != '\n' serve solo a svuotare il buffer di input nel caso l'utente non inserisca un numero in ingresso. Se l'utente seleziona un numero consono allora si procede alla write(), con la quale invieremo il valore alla scheda. Il valore di ritorno è il numero di byte scritti, quindi l'if seguente controlla che tutto sia andato a buon fine.

Ogni volta che l'utente seleziona delle opzioni, vengono eseguiti gli step che si possono eseguire con le nuove informazioni. Per esempio, dopo aver scoperto il numero di canali che verranno utilizzati si procede a scrivere sul file di output dei risultati una prima riga contenente il nome delle colonne. Dopo la selezione della modalità di esecuzione viene scelto il t_min. Questo viene calcolata per la modalità buffer, a partire dalla dimensione della SRAM della scheda e dal numero di canali ADC, mentre è stato trovato empiricamente nel caso della continous mode. Infine, dopo che l'utente ha selezionato il periodo di campionamento, questo è usato per calcolare il numero di conversioni che avverrà (max_conv) per ogni canale, la lunghezza totale dell'array che conterrà i dati, e quindi quanti byte dovrò leggere (len), e per ultimi vengono allocati i buffer che saranno utilizzati per leggere i dati (buffer) ed eventualmente per scriverli (buffer_out).

Una volta che il trigger è stato inviato, il codice si divide a seconda della modalità che è stata selezionata, per meglio rispondere alle loro esigenze.
Per la modalità continous, che richiede l'invio immediato dei dati, si va a sfruttare il periodo di stop che c'è tra la fine di una conversione su tutti i canali richiesti e quella successiva. Quindi, in pratica, si avrà che i dati verranno trasferiti entro il periodo di campionamento che è stato selezionato dall'utente, come

già ribadito al paragrafo precedente.
Essendo quindi a conoscenza del numero di byte che ci aspettiamo per ogni conversioni, andremo a leggere dalla scheda questo numero di byte per un numero di volte pari al numero di conversioni che accadranno (max_conv).
Dopo aver ricevuto i dati, questi vengono elaborati e stampati sul file di output in formato floating point, per poterli poi plottare con GNUPLOT. L'ADC restituisce numeri compresi tra 0 e 1023, per trasformarli nel valore di voltaggio letto ci rifacciamo al datasheet, che alla sezione "26.7 ADC Conversion Result" ci riporta la seguente formula:

For single ended conversion, the result is

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

Per avere il nostro risultato allora ci basta invertire la formula.

Per la buffered mode i valori vengono inviati tutti insieme, quindi si procederà a leggerli in blocchi, rielaborarli "appoggiandoli" nel vettore buffer_out e da questo scriverli sul file di output. La lettura avviene in blocchi di 64 byte per volta perchè più di 64 byte cominciavano ad apparire valori sbagliati e il programma non riusciva a proseguire, anche settando correttamente la variabile "tty.c_cc[VMIN]".

Infine, vengono deallocati i buffer e viene chiuso il file di output.

Per quanto riguarda i file serial_linux.c e serial_linux.h, questi sono stati presi dagli esempi fatti a lezione, modificando leggermente serial_linux.c. In particolare è stata aggiunto il baudrate di 38400, è stato "esplicitato" il comando "cfmakeraw(&tty);" per cercare l'errore nella lettura di blocchi di byte in numero maggiore di 64, ed è stata modificata la variabile "tty.c_cc[VMIN]", ponendola uguale a 64 per letture in modalità blocking.

2.3 GNUPLOT

Lo script per l'oscilloscopio contiene impostazioni di base per settare come output un file .png, per settare gli assi, sia come grandezze che come nomi, e infine contiene un grande if che a seconda del numero di canali che sono stati usati per la conversione va a plottare un numero appropriato di linee. Ho dovuto utilizzare un if visto che uno "switch" non esiste nella sintassi degli script di GNUPLOT.
La condizione contenuta nell'if prende il numero di canali passato dall'utente, ci somma zero per trasformarlo in un intero (invece di restare una stringa) e lo confronta con le opzioni disponibili.
Il comando necessario a far partire lo script è presente nello stesso commentato alla seconda riga, ma viene anche riportato nella sottosezione "Plottare i grafici con lo script per GNUPLOT" verso la fine della relazione, con la sua spiegazione.

3 Come compilare il progetto e Testarlo

Per compilare il programma bisogna prima di tutto individuare la porta a cui è collegata la scheda ed effettuare le scelte riguardanti il baudrate, l'utilizzo della modalità approssimata e il tempo totale di operazione.
Per individuare la porta, collegare la scheda al PC tramite USB e controllare il nome della porta a cui è stata attaccata. Di solito sarà la porta /dev/ttyACM0 ma potrebbe variare.
Una volta individuata, se diversa da ttyACM0, la porta corretta va indicata nei file:

- ./PC/main_pc.c riga 9
- ./AVR/avr_common/avr.mk riga 11

per quanto riguarda il baudrate invece, quello pre-impostato è 38400. Se si vuole operare con baudrate uguale a 19200 è necessario commentare la riga con "#define BAUD 38400" e decommentare quella con "#define BAUD 19200". Queste define si possono trovare in:

- ./PC/main_pc.c righe 7-8 per il programma PC
- ./AVR/avr_common/my_uart.h righe 5-6 per il programma per l'atmega

Per quanto riguarda la modalità approssimata, per selezionarla sarà necessario decommentare la riga "#define APPROX" che si può trovare in:

- ./PC/main_pc.c riga 13 per il programma PC
- ./AVR/avr_common/my_adc.h riga 4 per il programma per l'atmega

Infine per quanto riguarda il tempo totale di operazione, questo è pre-impostato a 10 secondi. Per cambiarlo basterà cambiare il valore della macro "STOP" che si può trovare in:

- ./PC/main_pc.c riga 10 per il programma PC
- ./AVR/main_avr.c riga 14 per il programma per l'atmega

e anche nello script gnuplot "oscilloscopio.plt" presente nella cartella ./PC, alla riga 12, come secondo valore dell' "xrange" in millisecondi.

3.1 Compilare e Caricare il programma per l'Atmega2560

Fatte le scelte indicate nel paragrafo precedente si può procedere alla compilazione. Per compilare il programma per l'atmega basterà andare nella cartella ./AVR/ e da terminale usare il comando "make". Per caricare il programma compilato sulla scheda il comando da usare è "make main_avr.hex":

```
1 $ make
2 $ make main_avr.hex
```

Se si cambia una delle macro tra BAUD e APPROX sarà necessario chiamare il comando "make clean" prima di effettuare il "make" per compilarlo.

3.2 Compilare ed Utilizzare il programma per PC

Per compilare il programma per PC basta andare nella cartella ./PC/ e digitare il comando "make".
Per utilizzare il programma lanciare l'eseguibile creato e passare come primo argomento il nome del file in cui verranno salvati i risultati, possibilmente con estensione ".dat". Quindi un esempio è:

```
1 $ make
2 $ ./main_pc Risultati.dat
```

3.3 Plottare i grafici con lo script per GNUPLOT

Una volta finita l'acquisizione dei dati dalla scheda, e quindi una volta che il programma per PC è arrivato alla sua chiusura, per plottare i risultati basterà usare il seguente comando quando ci si trova nella cartella ./PC/:

```
1 $ gnuplot -c oscilloscopio.plt NUMERO_CANALI_ADC_USATI NOME_FILE_RISULTATI.dat
```

Lo script genererà un file .png chiamato "oscilloscopio.png" contenente il plot dei dati acquisiti.