EMULATORE SRIX4K

PROOF OF CONCEPT

Ver 1.0 - 12/02/20 By Ptr

Alcune note che è meglio sottolineare:

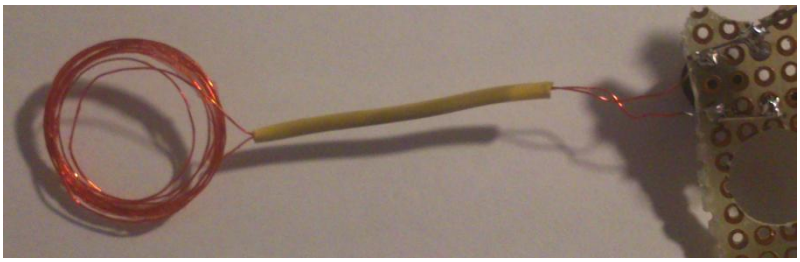
- È solo un "Proof Of Concept" di un emulatore, non un emulatore fatto e finito.
- L'emulatore non è stato testato "sul campo" ma solo in casa utilizzando una demo board della STmicroelectronics (anche un PN532 va bene).
- L'emulatore non è "completo". Solo i comandi *Initiate*, *Select chip ID*, *Read*, *Write* sono stati implementati. Non per sadismo o per una mia intenzione di non cedere chissà che segreti industriali (il solito concetto di "pappa pronta", ecc.) ma semplicemente perché non avevo voglia di implementare altri comandi per qualcosa che magari "sul campo" non funziona neanche. Alla fine, come ho detto, è un "Proof Of Concept".

Buon divertimento,

Ptr

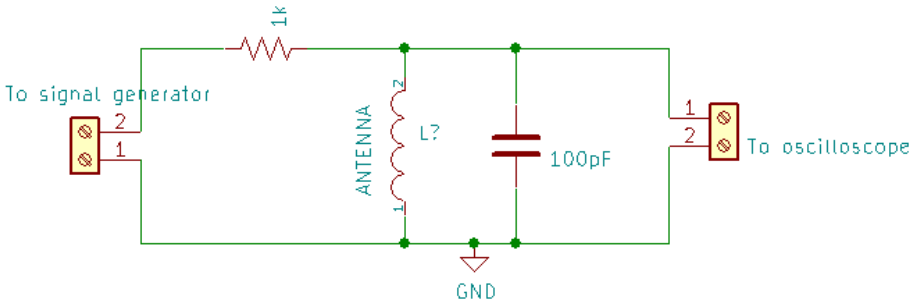
Antenna

Per realizzare l'antenna ho utilizzato 10 spire di filo di rame smaltato. Il diametro dell'antenna è di 1,5 cm.



Quel tubetto giallo è guaina termorestringente che ho utilizzato per raccordare l'antenna con la basetta millefori.

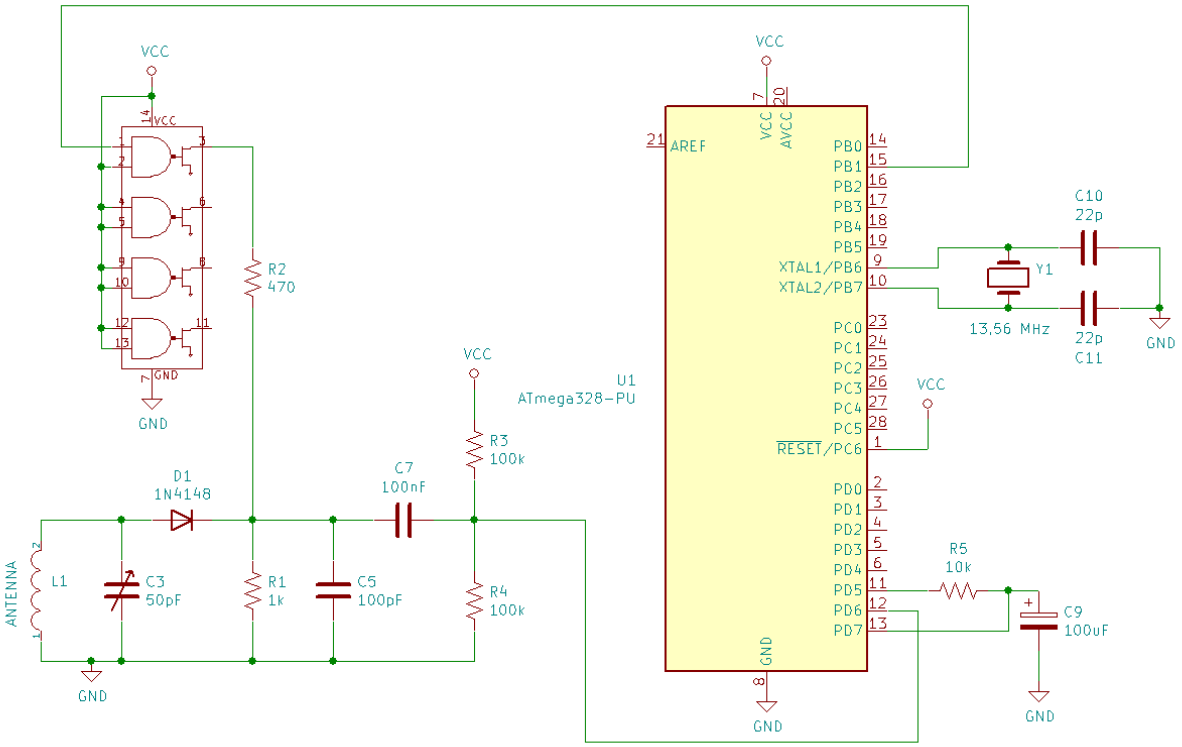
Per calcolare il valore di induttanza dell'antenna ho utilizzato questo circuito:



Si fa variare la frequenza del generatore di onda sinusoidale, alla risonanza la tensione di uscita visualizzata sull'oscilloscopio è massima.
Ho la risonanza a 8,5 MHz, il valore di induttanza è all'incirca 3,5uH.
Per avere risonanza a 13,56 MHz, il valore del condensatore deve essere circa 40pF. Utilizzo un trimmer capacitivo da 50pF per centrare la risonanza esattamente a 13,56 MHz.

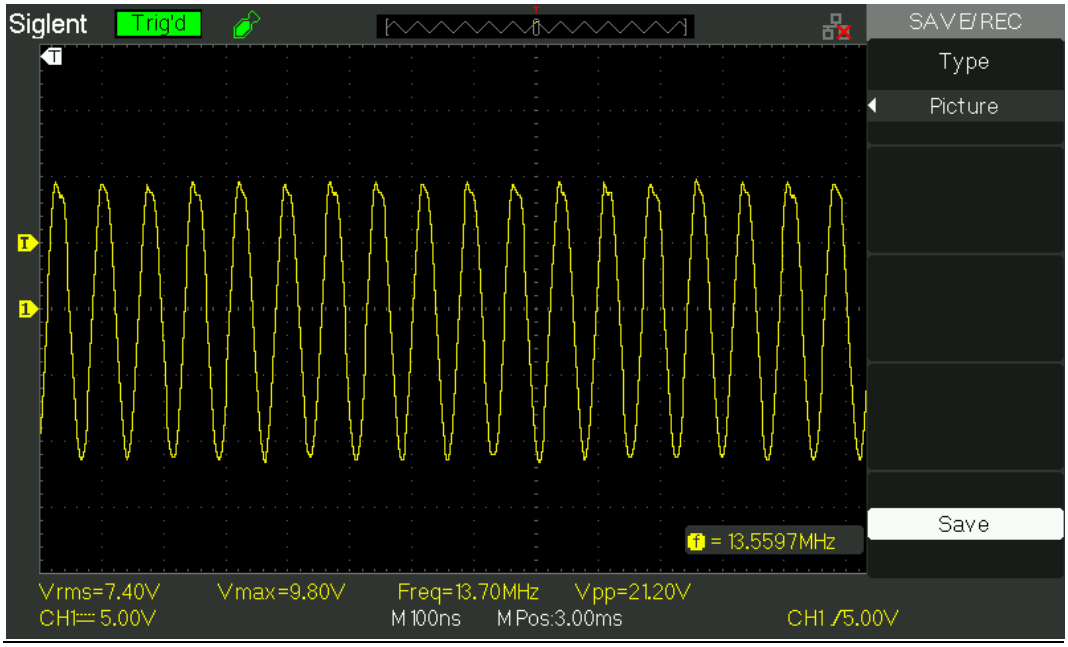
I trimmer capacitivi sono acquistabili qui: https://www.aliexpress.com/item/33041115782.html

Schema dell'emulatore

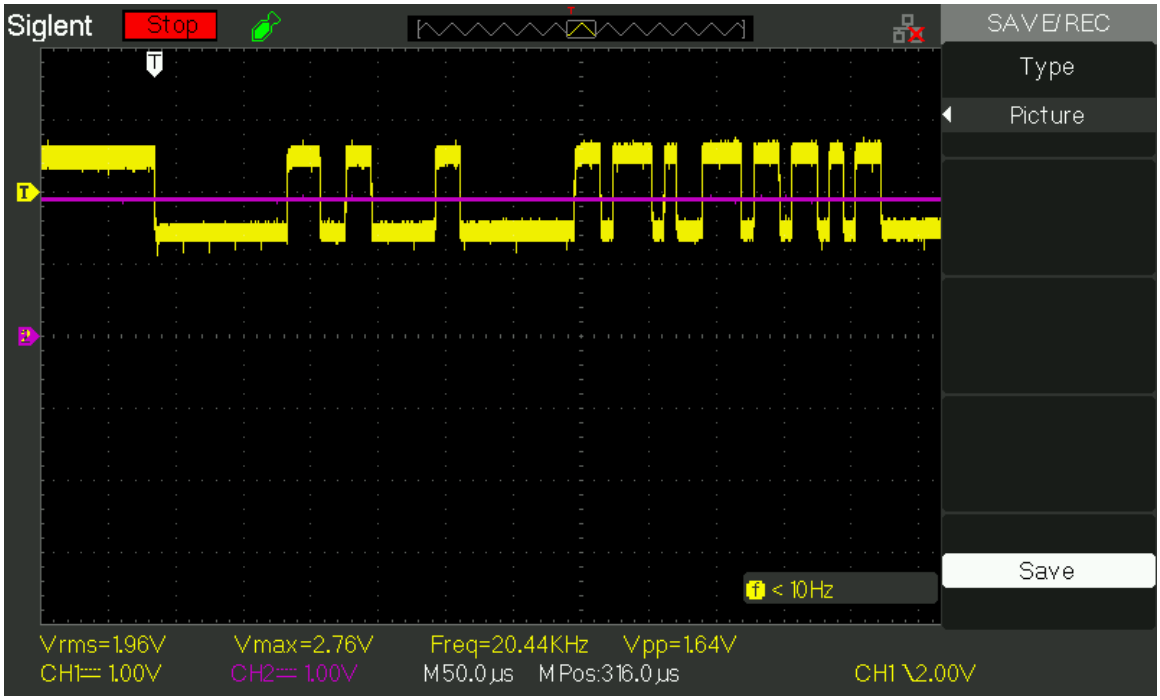


Il circuito è composto da:

- *L1/C3 - Circuito risonante*
  Il trimmer C3 deve essere tarato per avere la massima tensione sull'antenna quando questa è poggiata sul lettore.
  Nota: per i miei test ho usato la demo board della ST chiamata **M24LR-Discovery**. In questa board è sempre presente una portante a 13,56 MHz quando si seleziona la modalità ISO14443-B. Non so se le altre board tipo il PN532 fanno altrettanto (dovrei provare ma non ho voglia).
  Questa è uno screenshot della tensione sull'antenna:



- *D1/R1/C5 - Rivelatore di inviluppo*
  Come diodo rivelatore ho usato un 1N4148 perché è quello che avevo in casa, consiglio di usare un diodo schottky per avere una minor caduta di tensione su di esso.
  I valori di R1 e C5 sono stati ricavati tramite misure e varie prove sul circuito. Maggiore è la costante di tempo RC, maggiore sarà l'abilità del rivelatore di inviluppo di filtrare la portante in ingresso. Questo però causa dei problemi, perché oltre a filtrare la portante si rischia anche di filtrare il segnale utile che vogliamo ricavare! Difatti maggiore è la costante di tempo RC, maggiore è anche il filtraggio del segnale utile (modulante), per cui avrò in uscita un'onda quadra con i fronti molto lenti.
  La scelta di R1 e C5 è quindi un compromesso fra ripple in uscita e "rapidità" dei fronti del segnale.
  Ad esempio, con 100KΩ/100pF ho meno ripple a 13,56 MHz in uscita ma i fronti del segnale utile sono molto lenti. Con 1KΩ/100pF il ripple è maggiore ma il segnale utile è una bella onda quadra.
- *C7/R3/R4 - Rimozione dell'offset in continua*
  Il segnale in uscita al rivelatore di inviluppo presenta un offset in DC. Per questo motivo è stato utilizzato un condensatore per rimuovere questo offset e un partitore resistivo (R3 e R4) per portare l'offset a VCC/2.
  Anche questi valori sono stati ricavati facendo varie prove, di fatto C7 assieme ad R3 e R4 formano un filtro passa alto, se la frequenza di taglio di questo filtro è troppo alta c'è il rischio di filtrare anche il segnale utile che vogliamo misurare.
  L'uscita di questo stadio è collegata al pin 12 dell'ATMEGA328P che corrisponde all'ingresso non invertente del comparatore interno (Analog Comparator).
- *R5/C9 - Riferimento di tensione*
  Per ottenere un segnale digitale utile è necessario che sul pin non invertente del comparatore sia presente una tensione tale da poter effettuare una decisione corretta dei bit. A questo scopo vengono utilizzati R5/C9 per filtrare un segnale PWM a duty cycle variabile in uscita dall'ATMEGA.
  Sul pin 11 (configurato come uscita digitale) è presente un'onda quadra con un duty cycle tale da generare la tensione voluta su C9. Il segnale sul condensatore C9 viene riportato al pin 13 che corrisponde all'ingresso invertente del comparatore interno.
  Questo è uno screenshot delle due tensioni. In giallo è indicato il segnale al pin 12, in viola è indicato il riferimento di tensione generato da R5/C9 (il segnale al pin 13).



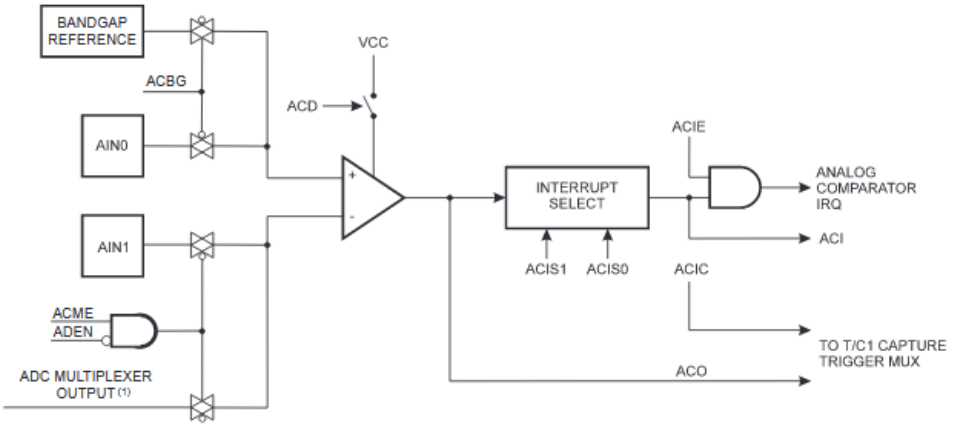
Come potete vedere, il riferimento di tensione è piazzato a metà dinamica del segnale digitale all'uscita del demodulatore. Questo riferimento è stato misurato facendo eseguire una serie di comandi al lettore di SRIX4K con l'antenna appoggiata sopra, in maniera tale da poter vedere dove piazzare il riferimento per avere una decodifica corretta.

- *R2 - Resistenza di modulazione*
  Il pin 15 viene usato per generare l'onda quadra BPSK in risposta ai comandi inviati dal lettore. Il pin finisce all'ingresso di una porta NAND (integrato CD74HC03E).
  Ho utilizzato questo integrato perché ha le uscite open drain. Praticamente pilotando l'ingresso si va a controllare il gate di un Mosfet che porta verso massa la resistenza R2 o lascia in alta impedenza il nodo. In questo modo vado a "caricare" il circuito LC generando una variazione del campo elettromagnetico emesso dal lettore.
  NOTA: L'idea iniziale era quella di usare un mosfet, purtroppo i mosfet di segnale che ho ordinato su Aliexpress non sono ancora arrivati, per questa versione mi sono arrangiato con quello che avevo. Quando mi arrivano vi faccio sapere, nel caso aggiorno lo schema.
  Nella primissima versione dell'emulatore avevo usato un JFET ma funziona male ed è concettualmente sbagliato.
- *Y1/C10/C11 - Oscillatore a 13,56MHz*
  L'ATMEGA viene utilizzato con un clock a 13,56 MHz. In questo caso non ci sono problemi con le varie frequenze in gioco.
  Il data transfer rate è a 106 Kb/s, cioè (13,56 MHz)/128
  La frequenza della BPSK è 847 kHz, cioè (13,56 MHz)/16

Demodulazione del segnale

Come indicato nella sezione precedente, viene utilizzato il comparatore interno dell'ATMEGA (Analog Comparator) per squadrare il segnale in ingresso. Lo schema del comparatore interno è il seguente:

Figure 23-1. Analog Comparator Block Diagram(2)

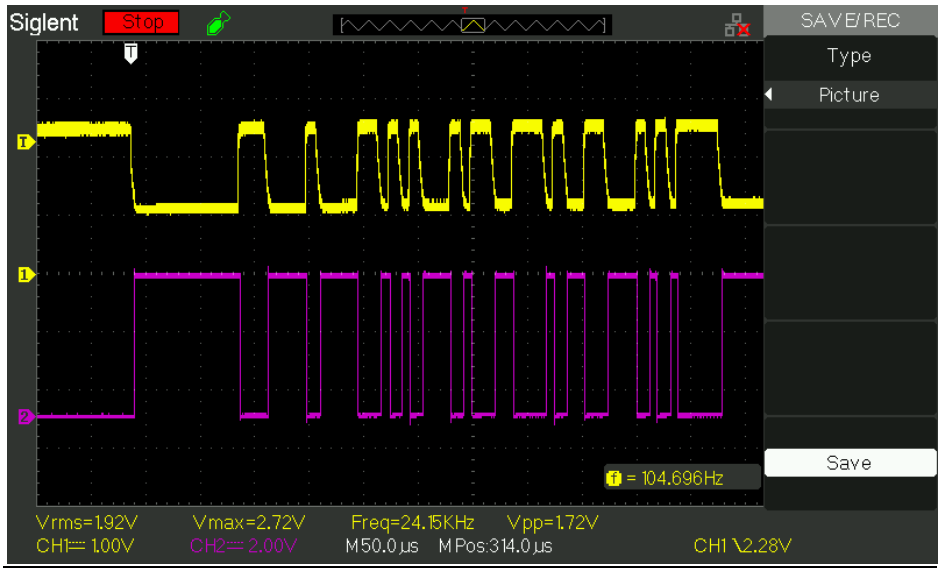


All'ingresso AIN0 (pin 12) è collegato il segnale in uscita dal partitore di R3/R4, all'ingresso AIN1 (pin 13) è collegata l'uscita del riferimento di tensione generato tramite R5/C9.
La scelta che ho fatto è stata quella di far generare un interrupt ogni volta che l'uscita del comparatore cambia di stato ("Comparator Interrupt on Output Toggle"):

Table 23-2. ACIS1/ACIS0 Settings

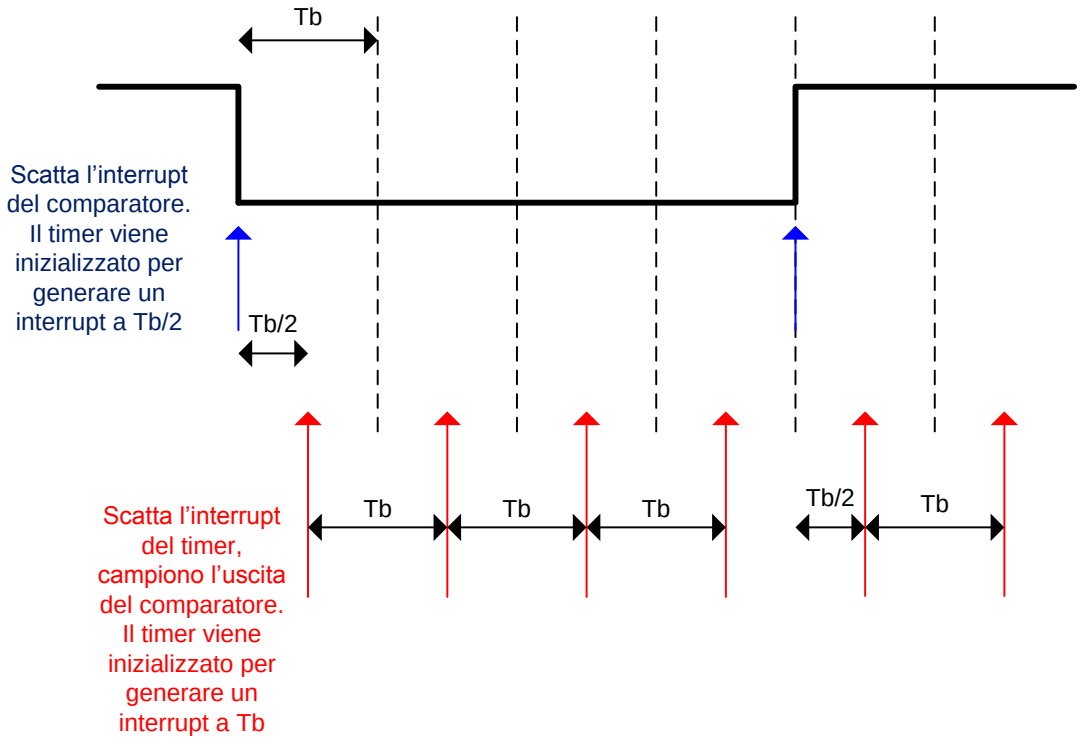
ACIS1	ACIS0	Interrupt Mode
0	0	Comparator Interrupt on Output Toggle.
0	1	Reserved
1	0	Comparator Interrupt on Falling Output Edge.
1	1	Comparator Interrupt on Rising Output Edge.

Per controllare che l'interrupt venisse generato correttamente, ho scritto un codice di prova che cambia lo stato di un bit su una porta di uscita ogni volta che la routine di interrupt viene eseguita. Questo è lo screenshot di ciò che si ottiene (il giallo è il segnale sul pin 12 e il viola è il bit della porta che cambia di stato) :



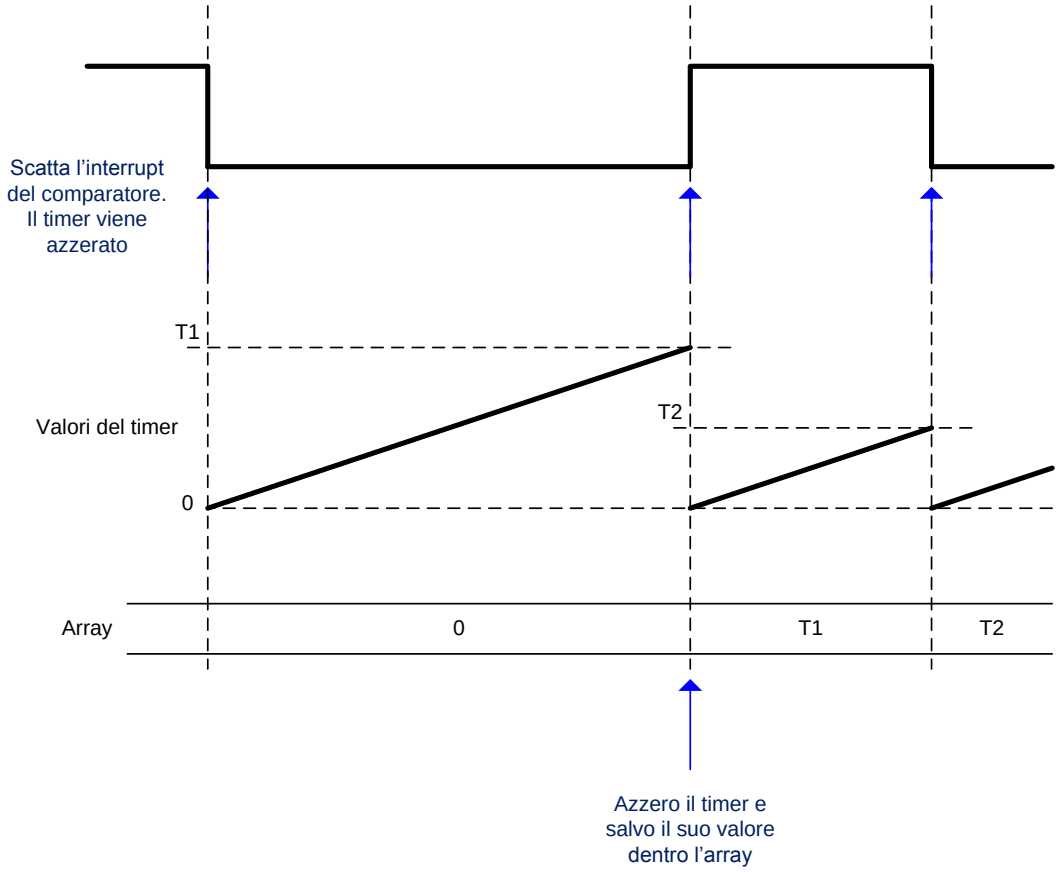
Come si può notare, l'interrupt viene generato correttamente ogni volta che ho una transizione del segnale in ingresso.

A questo punto ho provato questa strada: ad ogni transizione del segnale di ingresso (quindi ad ogni esecuzione della routine di interrupt) vado a resettare un timer e lo inizializzo in maniera tale da generare un interrupt dopo metà del tempo di bit (praticamente dopo (1/106000)/2 secondi).
Nella routine di interrupt del timer, leggo lo stato del comparatore e lo salvo in una variabile. A questo punto inizializzo il timer in maniera tale da generare un interrupt dopo un tempo di bit.
In questo modo vado a campionare lo stato del comparatore e mi "auto-sincronizzo" ad ogni fronte del segnale di ingresso.
Per capirci meglio faccio un disegno:



Spero che così sia più chiaro. Le frecce blu sono gli interrupt generati dal comparatore, le frecce rosse sono gli interrupt generati dal timer. Il nero è il segnale in ingresso sul pin 12.
Sulla carta questo sistema sembra perfetto, in pratica è una schifezza. Il campionamento dell'uscita del comparatore, l'inizializzazione dei timer e tutte le altre istruzioni dentro l'interrupt non sono a "tempo zero", hanno una loro durata e questo influenza le performance del sistema. Inoltre la forma d'onda del segnale di ingresso non è un'onda quadra perfetta per cui il campionamento delle volte avviene in maniera errata.
Io ho lasciato perdere questo sistema (a dire il vero non è che mi sono impegnato molto a cercare di migliorarlo in qualche modo), magari si può fare qualche ottimizzazione e funziona lo stesso. Sicuramente con una frequenza di clock maggiore si riesce a fare qualcosa di meglio.

Ho deciso quindi di utilizzare un altro sistema. Ogni volta che scatta l'interrupt del comparatore, viene azzerato un timer che parte a contare subito dopo. Al prossimo interrupt del comparatore, il valore del timer viene salvato in un array, il timer viene azzerato ed è pronto a contare l'intervallo di tempo successivo.
Per capirci meglio faccio un disegno:

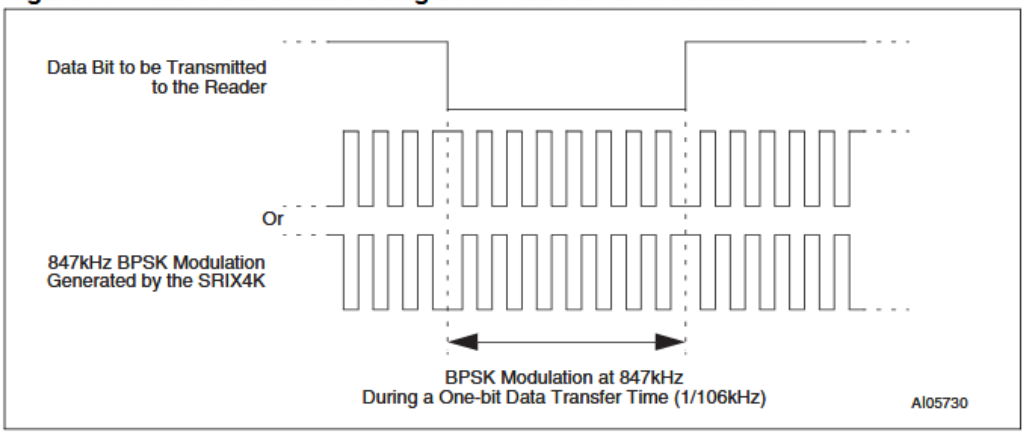


Praticamente memorizzo dentro l'array la durata di ogni transizione del segnale di ingresso. In questo modo è come se campionassi il segnale in ingresso con una risoluzione temporale pari a quella del clock di sistema a 13,56 MHz.
L'array, alla fine della ricezione del comando del lettore, deve essere elaborato per poter estrarre il bitstream utile che ci serve per la decodifica. Questo è purtroppo un collo di bottiglia del sistema, come vedremo più avanti.
Questo metodo comunque funziona bene ed è quello che è stato implementato nel codice.

Modulazione del segnale

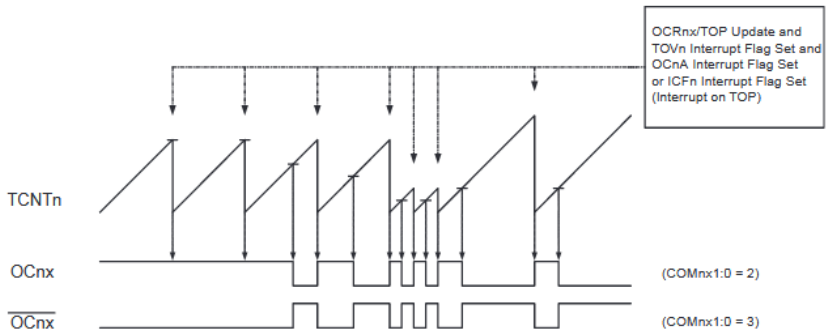
Per la generazione della risposta del tag, è necessario generare un segnale modulato BPSK, come descritto nel datasheet.

Figure 7. Wave transmitted using BPSK subcarrier modulation



Per la generazione di un'onda a 847 KHz (o meglio, 847,5 KHz come indicato in fondo al datasheet) è sufficiente utilizzare il timer nella modalità "Fast PWM".

Figure 16-7. Fast PWM Mode, Timing Diagram

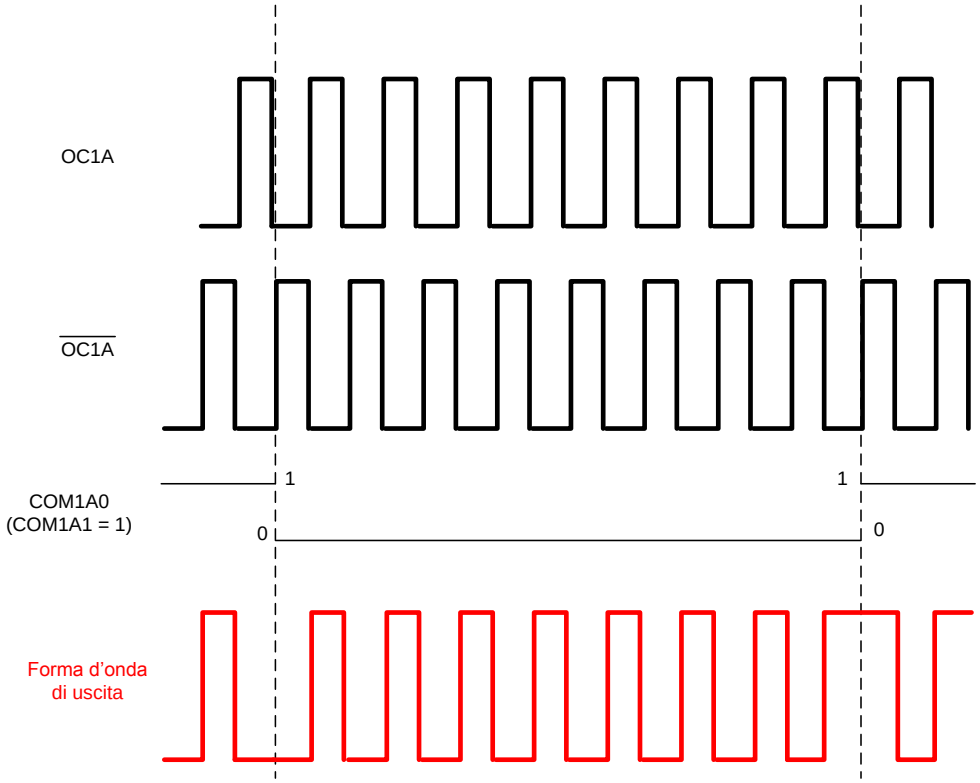


Utilizzando 15 come valore di TOP e 7 come valore di OCRA si riesce a generare un'onda quadra a (13,56 MHz)/16 = 847,5 KHz
Di per se la generazione dell'onda è semplice, il problema è la transizione di fase. Per fare questo, ho sfruttato la possibilità di invertire la fase dell'onda quadra che viene generata utilizzando i bit COMnx1:0 (vedi figura sopra).

Table 16-2. Compare Output Mode, Fast PWM(1)

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

Impostando COM1A1 a 1 e agendo su COM1A0 è possibile realizzare la transizione di fase che si vuole ottenere.



Per cambiare lo stato di COM1A0 si può utilizzare un altro timer (io utilizzo il timer 2) che genera un interrupt ogni (1/106000) secondi, praticamente ogni 8 cicli dell'onda a 847,5 KHz. All'interno della routine di interrupt si esegue l'istruzione che imposta lo stato di COM1A0 in base ai bit che si vuole trasmettere.
Questo sistema funziona solo se la transizione di COM1A0 è perfettamente in fase con il timer che genera l'onda quadra. Per fare ciò, utilizzo il registro GTCCR. Settando alcuni bit di questo registro è possibile fermare i timer, impostarli al valore che si vuole e poi farli ripartire. Maggiori informazioni sul funzionamento della sincronizzazione e del registro GTCCR le trovate qui:
http://www.openmusiclabs.com/learning/digital/synchronizing-timers/index.html

Gli step per generare la forma d'onda voluta sono i seguenti:

1. Fermo i timer.
2. Inizializzo il timer 1 per generare un'onda quadra a 847,5 KHz e il timer 2 per generare un interrupt ogni (1/106000) secondi.
3. Faccio ripartire i timer
4. Nella routine di interrupt del timer due cambio lo stato di COM1A0 in base ai bit che voglio trasmettere.

Ho realizzato un codice di esempio per testare la funzione di sincronizzazione (sync_example.c).
Questo codice serve ad illustrare la funzione di "cambio di fase" che si può ottenere cambiando il valore di COM1A0.
Nella funzione main abbiamo che:
Il pin PB1 (OC1A) viene settato come output, è il pin su cui abbiamo la forma d'onda di uscita a 847,5KHz.
Il pin PC0 viene settato come output e viene fatto cambiare di stato ogni volta che abbiamo un interrupt del timer 2:

```
//Set OC1A as output
DDRB = (1 << PB1);

//Set PC0 as output
DDRC = (1 << PC0);
```

Viene impostato il registro GTCCR per stoppare i timer dell'ATMEGA:

```
//Halt timers
GTCCR = (1 << TSM)|(1 << PSRASY)|(1 << PSRSYNC);
```

Viene impostato il timer 1 (generazione dell'onda quadra) e il timer 2 (interrupt ogni 1/106000 secondi).
L'interrupt del timer 2 viene fatto scattare quando OCR2A è uguale a (15*8)+7 = 127 (il data rate è 13,56 Mhz /128).

```
//Set timer one to Fast PWM (ICR1 = TOP)
ICR1 = 15;
OCR1A = 7;
TCCR1A = (1 << COM1A1) | (1 << WGM11);
TCCR1B = (1 << CS10) | (1 << WGM12) | (1 << WGM13);

//Set timer two to CTC mode (OCR2A = TOP)
OCR2A = (15*8)+7;
TIMSK2 = (1 << OCIE2A);
TCCR2A = (1 << WGM21);
TCCR2B = (1 << CS20);
```

Infine i contatori vengono resettati e fatti ripartire settando a zero il registro GTCCR:

```
//Reset all timer
TCNT0 = 0;
TCNT1 = 0;
TCNT2 = 0;

//Restart all timers
GTCCR = 0;
```

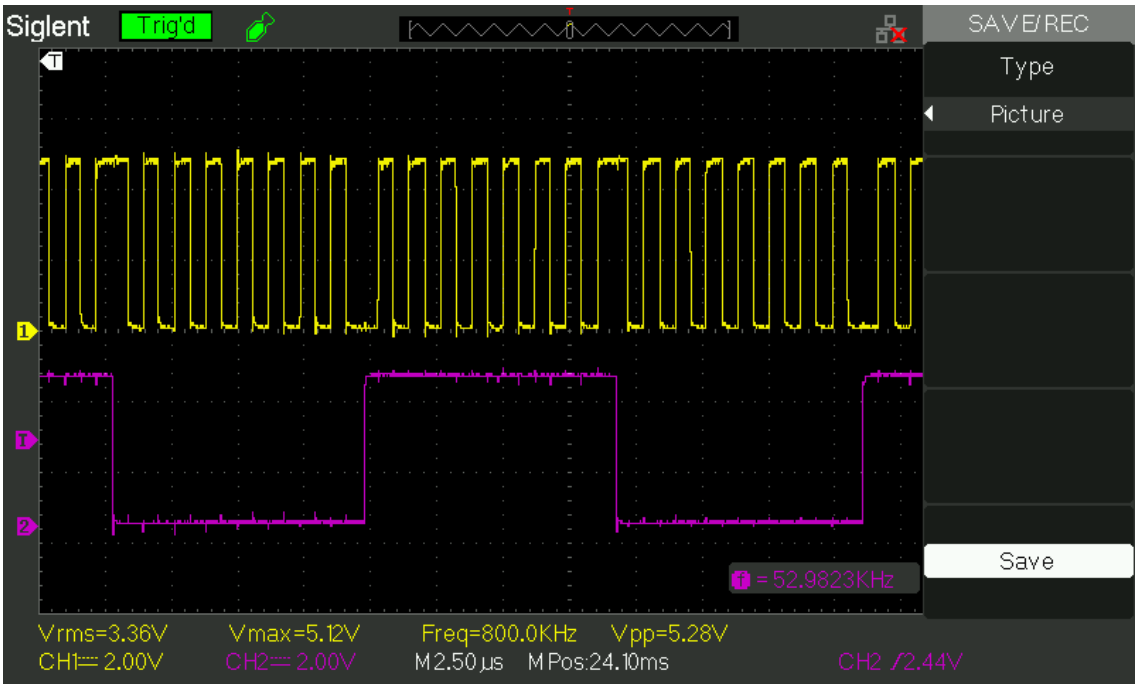
Nella routine di interrupt viene settato a uno o a zero il bit COM1A0 (è un bit del registro TCCR1A) in base allo stato della variabile **tx**, la quale cambia il suo valore (0/FF) ogni volta che l'interrupt viene eseguito. Lo stato di **tx** viene portato sul pin 0 della porta C:

```
unsigned char tx=0;

ISR(TIMER2_COMPA_vect)
{
    if (tx)
        TCCR1A = _BV(COM1A1) | _BV(WGM11);
    else
        TCCR1A = _BV(COM1A1) | _BV(COM1A0) | _BV(WGM11);

    tx = tx ^ 255;
    PORTC = tx & 1;
}
```

Ecco uno screenshot della forma d'onda a 847,5 KHz sul pin PB1 (traccia gialla) e del segnale che cambia ad ogni interrupt sul pin PC0 (traccia viola):



Come si può vedere, ogni volta che ho una transizione del pin PC0 (quindi ad ogni interrupt) avviene una transizione di fase della forma d'onda a 847,5 KHz (traccia gialla). È esattamente quello che vogliamo ottenere, a questo punto ci basta effettuare queste transizioni di fase in base al bitstream che vogliamo trasmettere.

Codice finale

In seguito vengono presentate alcune parti del codice dell'emulatore, in maniera tale da spiegarne a grandi linee il funzionamento.

Il codice è composto da quattro file:

- *crc.h*
  Contiene le funzioni per calcolare il CRC da utilizzare in trasmissione. Per semplicità (e per risparmiare cicli di clock), in ricezione non viene verificato che il CRC ricevuto sia corretto.
- *key.h*
  Contiene la variabile *mem*, utilizzata per rappresentare la memoria del tag, e la variabile *mem_FF* utilizzata per rappresentare l'indirizzo 0xFF della memoria.
- *serial.h*
  Contiene istruzioni base per la trasmissione di un dato via seriale. Nel codice non viene utilizzata, è stata lasciata solo per possibilità di debug.
- *main.h*
  Programma principale.

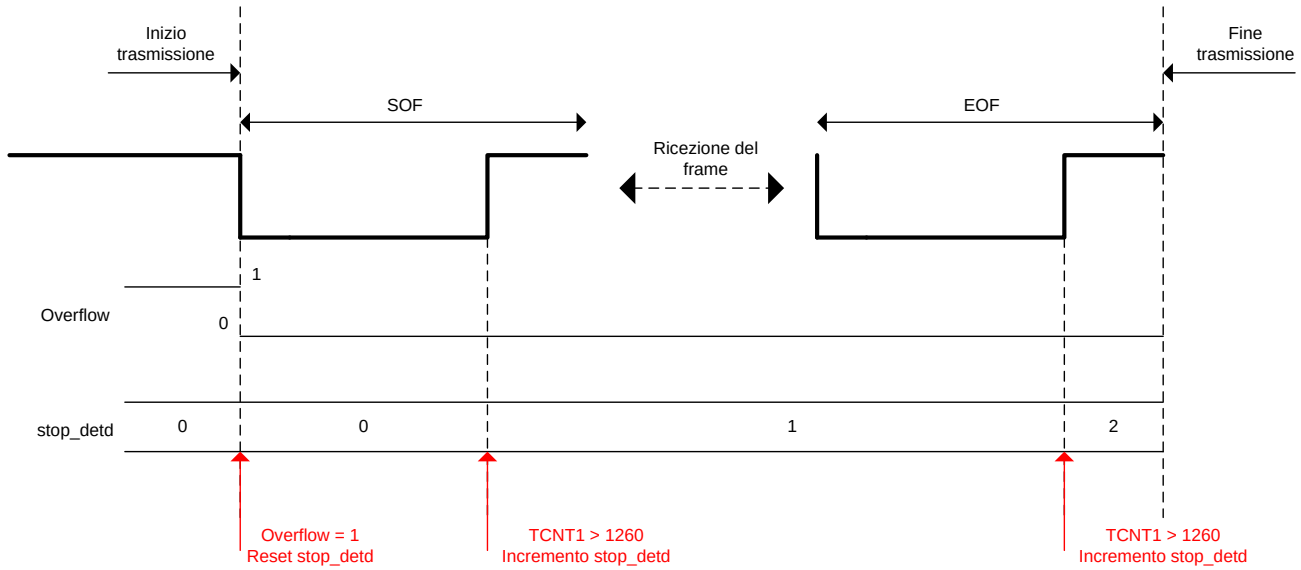
Per la ricezione del segnale, viene fatta eseguire la routine di interrupt del comparatore interno all'ATMEGA. Questa routine viene eseguita ogni volta che il segnale di ingresso compie una transizione 1-0 oppure 0-1.
Inizialmente, il contatore viene fermato e viene salvato il valore di esso all'interno di un vettore chiamato *raw*:

```
//ADC comparator interrupt
ISR(ANALOG_COMP_vect)
{
    //Stop the counter
    TCCR1B = 0;
    //Save data only if timer1 did not overflow
    if (!((TIFR1 >> TOV1) & 1))
        raw[raw_cnt++] = TCNT1;
```

Per stabilire la fine di una trasmissione valida viene utilizzata la variabile *stop_detd*. Questa variabile viene resettata ogni volta che ho un overflow del timer (l'overflow avviene sempre fra la ricezione di un comando e quello successivo) e viene incrementata ogni volta che il valore del timer è maggiore di 1260, cioè ogni volta che vengono ricevuti i 10 bit a zero del SOF e del EOF (maggiori informazioni più avanti):

```
    //Reset stop counter when overflow
    if ((TIFR1 >> TOV1) & 1)
        stop_detd = 0;
    else if (TCNT1>1260)
        stop_detd++;
```

In seguito viene mostrato un grafico per capire meglio il funzionamento di *stop_detd*.



La trasmissione possiamo considerarla conclusa quando *stop_detd* è uguale a due.
La routine di interrupt finisce pulendo il flag di overflow, resettando il contatore e facendo ripartire il timer per una nuova acquisizione.

```
    //Clear overflow flag
      TIFR1 = (1 << TOV1);
    //Reset the counter
      TCNT1 = 0;
    //Start the counter
      TCCR1B = (1 << CS10);
}
```

La variabile *stop_detd* viene controllata nella funzione main. Ogni volta che *stop_detd* risulta essere maggiore o uguale a due, il contatore viene fermato, l'interrupt del comparatore viene disabilitato e viene richiamata la funzione **decode_bits**. Questa funzione effettua la "traduzione" fra intervalli di tempo e bit del frame trasmesso:

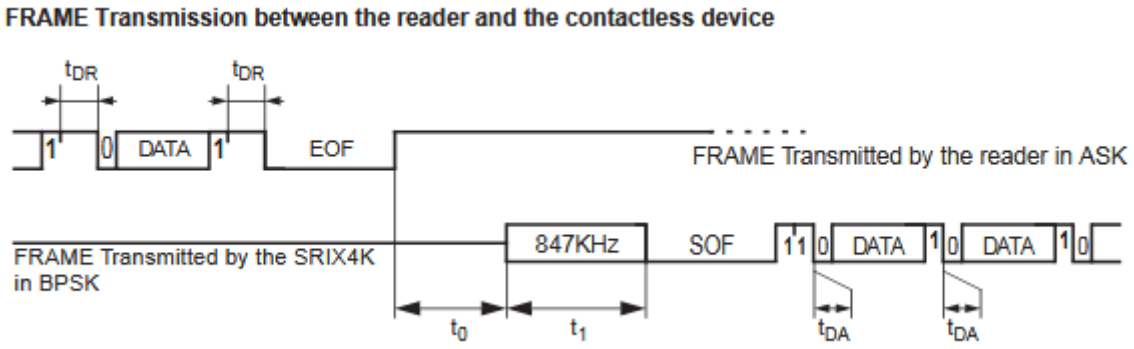
```
while (1)
{
    if (stop_detd >= 2)
    {
        //Stop Timer1 counter
        TCCR1A = 0;
        TCCR1B = 0;
        //Clear Timer1 overflow interrupt
        TIFR1 = (1 << TOV1);
        //Reset Timer1
        TCNT1 = 0;

        //Clear comparator interrupt flag
        ACSR = (1 << ACI);

        //Clear stop counter flag
        stop_detd = 0;

        if (decode_bits())
        {
```

La funzione **decode_bits** è una funzione particolarmente critica. Deve tradurre gli intervalli di tempo misurati e preparare un array con i dati da trasmettere. La funzione utilizza molti cicli di clock per svolgere le operazioni che deve effettuare.
Dal datasheet dello SRIX4K è indicato il tempo minimo che intercorre tra la fine di una ricezione dati dal lettore e un inizio della trasmissione dati da parte del tag. Il tempo si chiama t_0 ed è indicato come "Antenna reversal delay" (151 us):



Oltre a un tempo minimo esiste anche un tempo massimo, stabilito dal lettore. Quest'ultimo non può stare ad aspettare una risposta dal tag per un tempo infinito, ha un suo timeout interno. Questo timeout cambia da lettore a lettore.
Il tempo impiegato dalla funzione **decode_bits** è di circa 250us. Siamo ben oltre il tempo minimo, ma in ogni caso il mio lettore riesce comunque a ricevere correttamente i dati trasmessi.
Questo è comunque uno dei problemi di questo sistema. Ho cercato di fare il possibile per ottimizzare la funzione, margini di miglioramento ce ne sono ed è su questo che si deve lavorare.

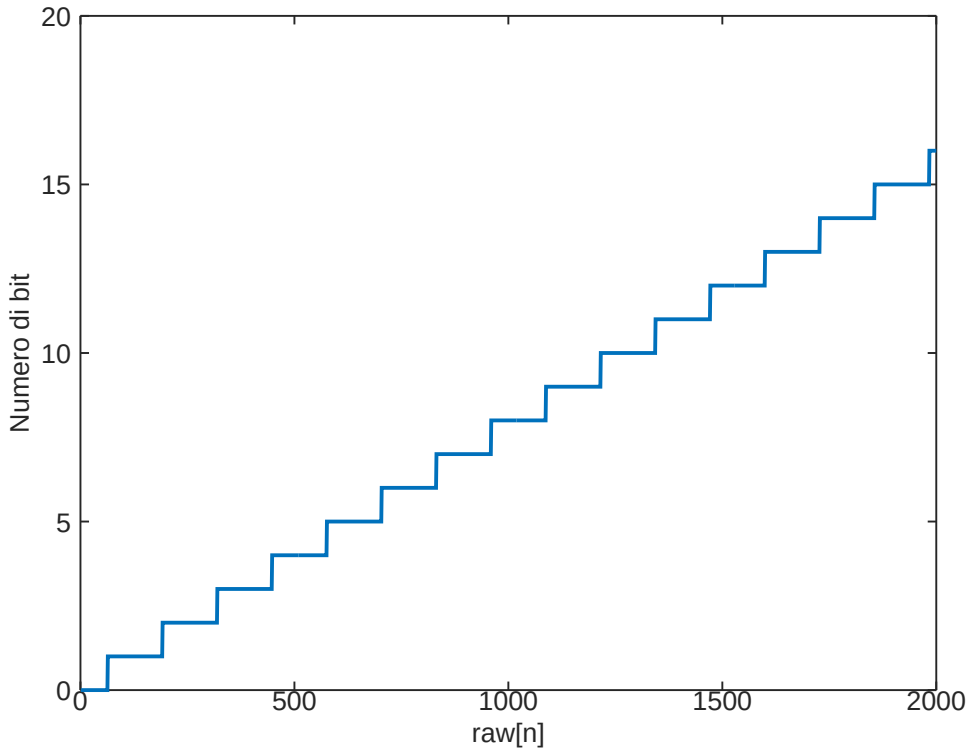
Per ricavare il numero di bit in un determinato intervallo di tempo dobbiamo:

- Moltiplicare il valore del contatore salvato nell'array per (1/13.56 MHz). In questo modo otteniamo il tempo effettivo in secondi dell'intervallo di tempo misurato. Il timer non ha prescaler per cui ogni "tick" è un colpo di clock a 13,56 MHz.
- Dividere l'intervallo di tempo ottenuto per il tempo di bit (128/13.56 MHz). In questo modo ricaviamo quanti bit abbiamo ricevuto in un determinato intervallo di tempo.
- Usare la funzione ROUND per effettuare un arrotondamento del numero reale ottenuto.

Scritto in formula:

$$Numero\ di\ bit = ROUND\left(\frac{raw[n]}{13560000} \times \frac{13560000}{128}\right) = ROUND\left(\frac{raw[n]}{128}\right)$$

Dove n è l'indice del vettore che contiene i valori del timer che sono stati memorizzati.
Dobbiamo realizzare una funzione che ha come ingresso i valori del timer e ha come uscita il numero di bit. Invece di effettuare una divisione e utilizzare la funzione ROUND, possiamo implementare questa funzione come se fosse una look-up table. Usando Excel o qualsiasi altro software, possiamo calcolare il numero di bit facendo variare il dato raw[n] da 0 a 2000.



A questo punto la funzione può essere implementata con degli if-else:

```
//Implement rounding function with a lookup table
unsigned char find_bit(unsigned int val)
{
    if ((val>=60) && (val<192))
        return 1;
    else if ((val>=192) && (val<320))
        return 2;
    else if ((val>=320) && (val<448))
        return 3;
    else if ((val>=448) && (val<576))
        return 4;
    else if ((val>=576) && (val<704))
        return 5;
    else if ((val>=704) && (val<832))
        return 6;
    else if ((val>=832) && (val<960))
        return 7;
    else if ((val>=960) && (val<1088))
        return 8;
    else if ((val>=1088) && (val<1216))
        return 9;
    else if ((val>=1216) && (val<1344))
        return 10;
    else if ((val>=1344) && (val<1472))
        return 11;
    else if ((val>=1472) && (val<1600))
        return 12;
    else
        return 0;
}
```

Idea dell'ultim'ora:
Riguardando la formula ho notato che la divisione per 128 si ottiene semplicemente shiftando a destra di 7 bit il dato raw[n]. Il ROUND si può realizzare guardando i bit che vengono troncati....
Vi faccio sapere.

La funzione **decode_bits** è abbastanza lunga e "contorta", non ho intenzione di spiegarla troppo nel dettaglio (alla fine quello che fa è tradurre i valori del contatore in bit per poi assemblarli assieme e formare i dati che il lettore ha inviato).

Gli step principali sono:

- Riconoscere l'inizio del comando, andando a ricavare la posizione nel vettore in cui inizia la trasmissione dati vera e propria, dopo l'invio dell'SOF. Un SOF è composto da un intervallo di tempo in cui vengono trasmessi dieci bit e un altro intervallo in cui ne vengono trasmessi due. Nel primo do-while della **decode_bits** viene fatto questo.
- Estrazione dei comandi ricevuti a partire dai valori salvati dentro il vettore *raw*.

In seguito vi mostro un esempio di una ricezione di un comando "Initiate":

	Valore	**Numero di bit**	**Bitstream**
raw[0]	1293	10	0000000000
raw[1]	271	2	11
raw[2]	206	2	00
raw[3]	211	2	11
raw[4]	593	5	00000
raw[5]	211	2	11
raw[6]	1102	9	000000000
raw[7]	213	2	11
raw[8]	78	1	0
raw[9]	339	3	111
raw[10]	78	1	0
raw[11]	84	1	1
raw[12]	206	2	00
raw[13]	341	3	111
raw[14]	78	1	0
raw[15]	211	2	11
raw[16]	79	1	0
raw[17]	211	2	11
raw[18]	79	1	0
raw[19]	83	1	1
raw[20]	79	1	0
raw[21]	211	2	11
raw[22]	1295	10	0000000000

Riportando tutti i bit in fila:

0000000000110011000001100000000110111010011101101101010110000000000

Si possono ricavare i seguenti dati:

000000000011 0 01100000 11 0 00000000 11 0 11101001 11 0 11011010 11 0000000000

- 000000000011
  Start Of Frame
- 0 01100000 11
  0 = Start
  01100000 = 0x06 (ricordatevi che il bit più significativo in realtà è l'LSB, quindi il dato va letto al contrario).
  1 = Stop
  1 = 1 ETU
- 0 00000000 11
  0 = Start
  00000000 = 0x00
  1 = Stop
  1 = 1 ETU
- 0 11101001 11
  0 = Start
  11101001 = 0x97
  1 = Stop
  1 = 1 ETU
- 0 11011010 11
  0 = Start
  11011010 = 0x5B
  1 = Stop
  1 = 1 ETU
- 0000000000
  End Of Frame. Mancano i due bit finali a 11 perché non vengono salvati nel vettore.

I dati trasmessi dal lettore sono quindi 0x06 0x00 0x97 0x5B, cioè il comando "Initiate" (0x06 0x00) con i due byte del CRC aggiunti alla fine.

NOTA: Mi è stato segnalato che alcuni lettori non inviano lo spazio di 1 ETU fra i comandi. Verificate quindi se il vostro lettore li invia, altrimenti la routine di decodifica non funzionerà (sono necessarie modifiche).

Per estrarre i bit la funzione **decode_bits** fa questa cosa:

```
res = find_bit(raw[n]);
temp = (temp >> res) + (bit_v*((1<<res) - 1) << (10-res));
```

La variabile *temp* andrà a contenere il frame estratto dalla durata dei bit salvata nel vettore *raw*. Come si è visto sopra, ciascun frame lo considero composto da 11 bit, cioè il bit di START, gli otto bit di informazione ("Information byte"), il bit di STOP e 1 ETU.
La variabile *res* contiene il numero dei bit estratto dal vettore *raw*, *bit_v* è una variable che viene fatta togglare 0-1 ogni volta che viene analizzato un nuovo elemento del vettore (ogni volta che n viene incrementato).
In seguito vengono elencati i vari passaggi della routine. Sperando di non creare troppa confusione...

	Valore	**res**	**bit_v**	**(bit_v*((1<<res) - 1)**	**(10-res)**	**$temp_{old}$ >> res**	**temp**
Inizio							0
raw[2]	206	2	0	0	8	0	0
raw[3]	211	2	1	11	8	0	1100000000
raw[4]	593	5	0	0	5	0000011000	0000011000
raw[5]	211	2	1	11	8	0000000110	1100000110
2 + 2 + 5 + 2 = 11 -> Primo frame ricevuto Per ottenere l'information byte -> temp & 255 = 00000110 = 0x06							
							0
raw[6]	1102	9	0	0	1	0	0
raw[7]	213	2	1	11	8	0	1100000000
9 + 2 = 11 -> Secondo frame ricevuto Per ottenere l'information byte -> temp & 255 = 00000000= 0x00							
							0
raw[8]	78	1	0	0	9	0	0
raw[9]	339	3	1	111	7	0	1110000000
raw[10]	78	1	0	0	9	0111000000	0111000000
raw[11]	84	1	1	1	9	0011100000	1011100000
raw[12]	206	2	0	0	8	0010111000	0010111000
raw[13]	341	3	1	111	7	0000010111	1110010111
1 + 3 + 1 + 1 + 2 + 3 = 11 -> Terzo frame ricevuto Per ottenere l'information byte -> temp & 255 = 10010111 = 0x97							
							0
raw[14]	78	1	0	0	9	0	0
raw[15]	211	2	1	11	8	0	1100000000
raw[16]	79	1	0	0	9	0110000000	0110000000
raw[17]	211	2	1	11	8	0001100000	1101100000
raw[18]	79	1	0	0	9	0110110000	0110110000
raw[19]	83	1	1	1	9	0011011000	1011011000
raw[20]	79	1	0	0	9	0101101100	0101101100
raw[21]	211	2	1	11	8	0001011011	1101011011
1 + 2 + 1 + 2 + 1 + 1 + 1 + 2 = 11 -> Quarto frame ricevuto Per ottenere l'information byte -> temp & 255 = 01011011 = 0x5B							

Una volta che vengono ricevuti i vari information byte, viene eseguita la funzione **decode_cmd** che interpreta il comando ricevuto e crea un vettore che contiene i dati che verranno trasmessi.
Ad esempio, per il comando "Initiate", la risposta da trasmettere viene creata così:

```
case(0x06):      //Initiate

data_tx[0] = 4095;
data_tx[1] = 255;
data_tx[2] = 768;
data_tx[3] = (17  + 256) << 1;
data_tx[4] = (112 + 256) << 1;
data_tx[5] = (241 + 256) << 1;
data_tx[6] = 0;
data_tx[7] = 3;

MAX_TX = 7;

break;
```

I dati da trasmettere vengono inseriti in un vettore chiamato *data_tx*, la variabile *MAX_TX* contiene il numero di elementi di questo vettore .
Se i dati in decimale vengono espressi in binario (espressi su 10 bit), si ottiene questo:

```
data_tx[0] = 1111111111;
data_tx[1] = 0011111111;
data_tx[2] = 1100000000;
data_tx[3] = 1000100010;   00010001 = 17
data_tx[4] = 1011100000;   01110000 = 112
data_tx[5] = 1111100010;   11110001 = 241
data_tx[6] = 0000000000;
data_tx[7] = 0000000011;
```

Per trasmettere questi dati viene utilizzata la routine di interrupt del timer 2, come spiegato nella sezione di Modulazione del segnale. Non starò a spiegare nel dettaglio la routine, è molto semplice.
Il vettore *data_tx* viene percorso dal basso verso l'alto e i bit vengono trasmessi dal meno significativo al più significativo. Se vado a "srotolare" il vettore ottengo questo:

1111111111 1111111100 0000000011 0100010001 0000011101 0100011111 0000000000 1100000000

- 1111111111 11111111
  Questa sequenza serve al lettore per sincronizzarsi con il tag (nel datasheet è definito come sync). In questo intervallo di tempo non c'è nessuna transizione di fase.
- 000000000011
  SOF
- 0100010001
  0 = Start
  10001000 = 17 in binario con LSB first
  1 = Stop
- 0000011101
  0 = Start
  00001110 = 112 in binario con LSB first
  1 = Stop

- 0100011111
  0 = Start
  10001111 = 241 in binario con LSB first
  1 = Stop
- 000000000011
  EOF

La trasmissione deve iniziare sempre con questi tre dati che corrispondono al SYNC + SOF:

```
data_tx[0] = 4095;
data_tx[1] = 255;
data_tx[2] = 768;
```

E concludersi sempre con questi due dati

```
data_tx[6] = 0;
data_tx[7] = 3;
```

In questo esempio, i dati da trasmettere in risposta al comando "Initiate" sono:

- 17 (0x11) = Chip ID
- 112 (0x70) = CRCL
- 241 (0xF1) = CRCH

Come ultima cosa, riporto la configurazione del timer 0, utilizzato per generare l'onda quadra sul pin 11, in maniera tale da generare un riferimento di tensione.

```
void setup_timer0()
{
    //Set OC0B on compare match, clear OC0B at BOTTOM
    TCCR0A = (1 << COM0B1) | (1 << COM0B0) | (1 << WGM01) | (1 << WGM00);
    //No prescaler
    TCCR0B = (1 << CS00);
    OCR0B  = 160; //Seems OK
    TIMSK0 = 0;
    TCNT0 = 0;
}
```

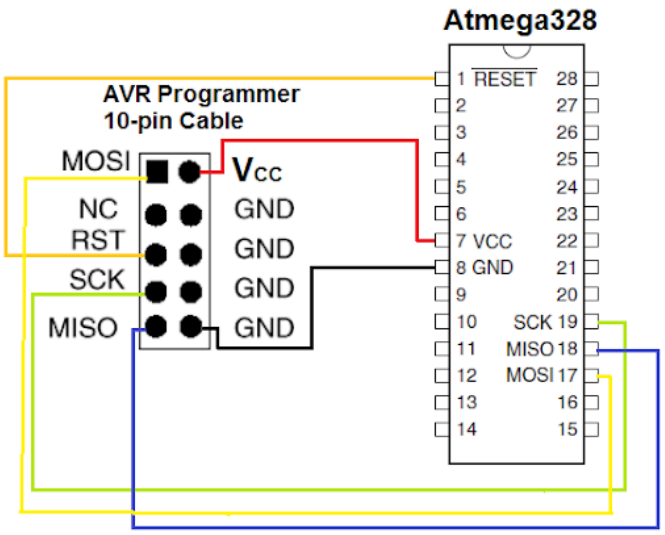
Non viene usato nessun prescaler, la frequenza di uscita è (13,56 MHz / 256) = 53 KHz.
Il valore del duty cycle si cambia agendo su OCR0B, il valore corretto è stato trovato a tentativi osservando la tensione sull'oscilloscopio.
I valori di R5 e C9 per il filtraggio dell'onda quadra sono forse eccessivi, credo che sia possibile utilizzare una costante di tempo più bassa e avere comunque una tensione stabile in uscita (anche un condensatore da 1 uF va bene lo stesso).

Miglioramenti possibili

- Ricavare VCC direttamente dall'antenna senza usare un'alimentazione esterna.
- Impostare il duty cycle per la forma d'onda su R5 e C9 in maniera automatica (es. si parte con duty cycle a 0 e lo si fa crescere fino a che non si riceve un comando valido).
- Riscrivere la routine di decodifica del comando inviato dal lettore in assembly per migliorare le prestazioni.
- Usare un altro metodo per la decodifica dei comandi inviati dal lettore.
- Altro?

Compilazione e scrittura sull'ATMEGA

Per la compilazione e la scrittura sull'ATMEGA vengono utilizzati WINAVR (scaricabile qui https://sourceforge.net/projects/winavr/) e il programmatore USBASP, acquistabile su internet a pochi euro.
Il collegamento del programmatore all'ATMEGA avviene in questo modo:



Per compilare il codice, utilizzare l'istruzione seguente al prompt dei comandi di Windows:

```
avr-gcc -Wall -g -Os -mmcu=atmega328p -std=gnu99 -o main.bin main.c
```

Per generare il file .hex

```
avr-objcopy -j .text -j .data -O ihex main.bin main.hex
```

Prima di scrivere il file .hex è necessario scrivere i fuse bit (NOTA: questa operazione è da fare una volta sola, non si deve ripetere ogni volta che si scrive il .hex)

```
avrdude -p atmega328p -c usbasp -U lfuse:w:0xEE:m
avrdude -p atmega328p -c usbasp -U hfuse:w:0xD9:m
```

Per scrivere il file .hex nel microcontrollore:

```
avrdude -p atmega328p -c usbasp -U flash:w:main.hex:i -F -P usb
```

Per chi ha linux, può scaricare tutti i tool usando il comando

```
sudo apt-get install gcc-avr avr-libc avrdude
```

Le istruzioni per la compilazione e la scrittura sono le stesse usate su windows.