EMULATORE SRIX4K

PROOF OF CONCEPT

Ver 2.1 - 26/02/22 By Ptr

Alcune note che è meglio sottolineare:

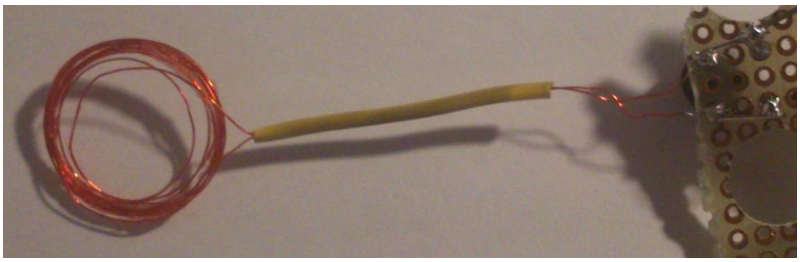
- È solo un "Proof Of Concept" di un emulatore, non un emulatore fatto e finito.
- ~~L'emulatore è stato testato "sul campo" ed è funzionante.~~
- Nel codice sono stati implementati i comandi base *Initiate*, *Select chip ID*, *Read*, *Write, Get_UID*, *Pcall16, Slot_marker*

Buon divertimento,

Ptr

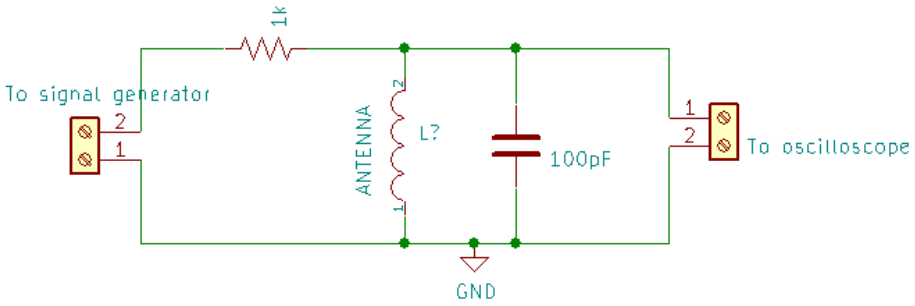
Antenna

Per realizzare l'antenna ho utilizzato 8 spire di filo di rame smaltato. Il diametro dell'antenna è di 1,5/1,8 cm.



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

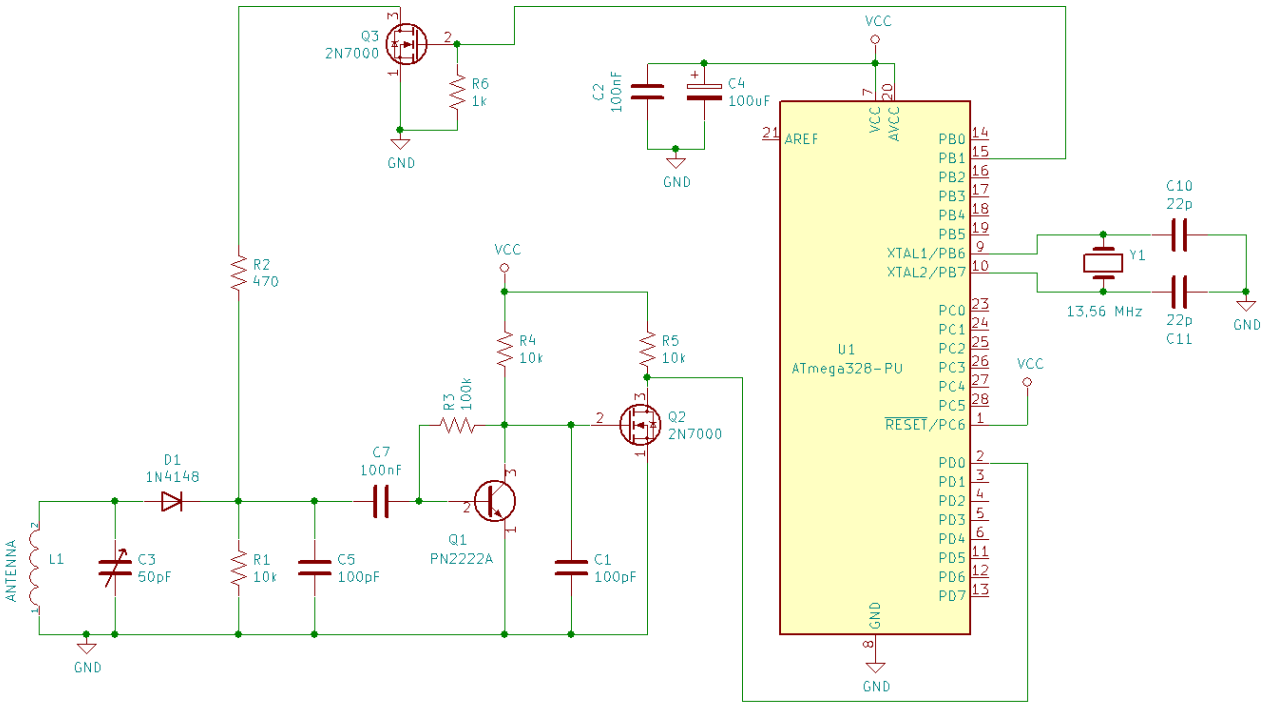
Per calcolare il valore di induttanza dell'antenna ho utilizzato questo circuito:



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 circa 9 MHz, il valore di induttanza è all'incirca 3uH.
Per avere risonanza a 13,56 MHz, il valore del condensatore deve essere circa 46pF. Utilizzo un trimmer capacitivo da 60pF per centrare la risonanza esattamente a 13,56 MHz.

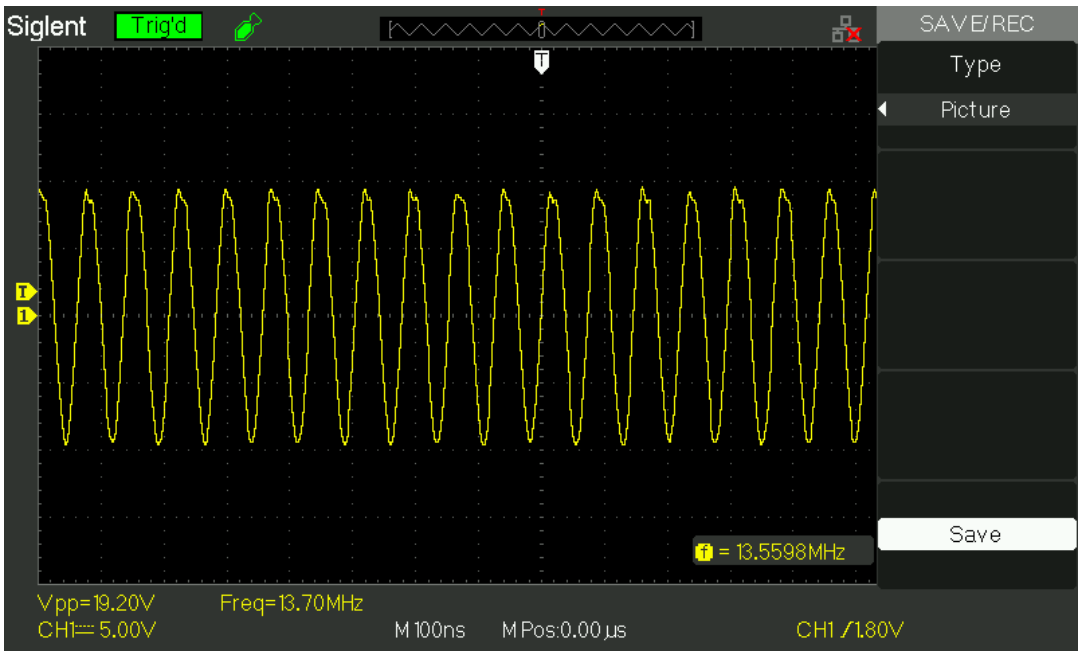
I trimmer capacitivi sono acquistabili qui: https://www.aliexpress.com/item/33041115782.html
Oppure qui: https://www.rf-microwave.com/en/murata/tz03p600er169/variable-capacitor-trimmer-9-8-60pf-100v/cvc-59/

Schema dell'emulatore

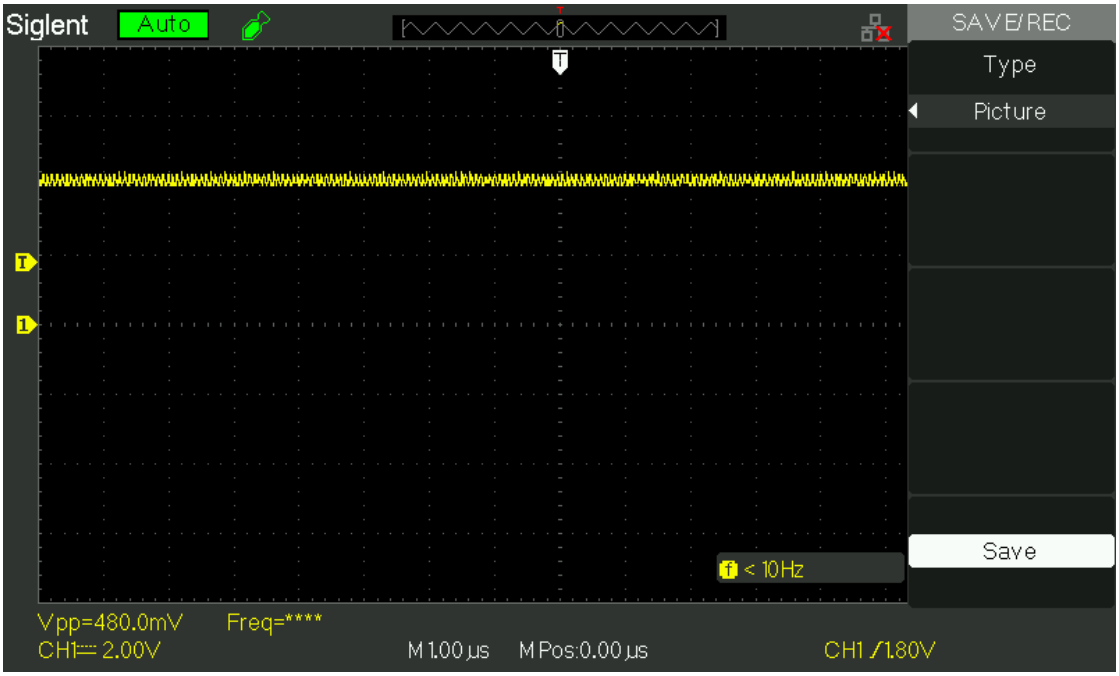


Il circuito è composto da:

- *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. Se utilizzate il PN532, usate lo sketch di Arduino e fatelo partire una volta senza avvicinare nessun tag.
  Questo è uno screenshot della tensione sull'antenna:

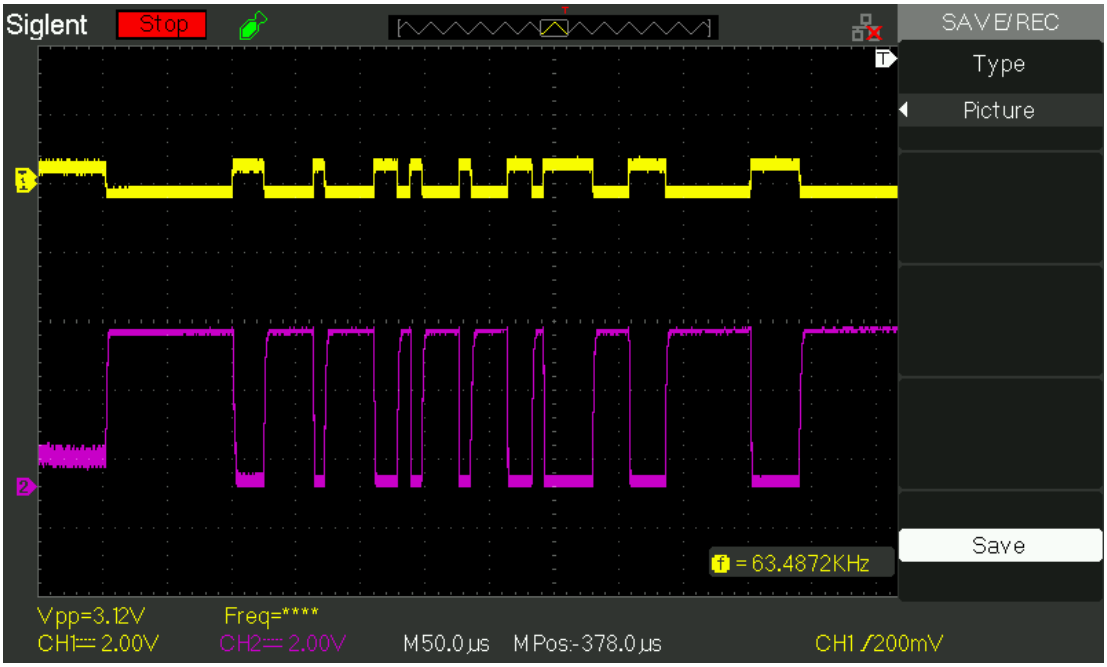


- *D1/R1/C5 - Rivelatore di inviluppo*
  Come diodo rivelatore ho usato un 1N4148 perché è quello che avevo in casa, consiglio di usare un diodo schottky per avere una minor caduta di tensione su di esso. Non usate diodi schottky "di potenza" tipo l'1N5819 (parlo per esperienza personale).
  I valori 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 10KΩ/100pF il ripple è maggiore ma il segnale utile è un'onda quadra con i fronti più ripidi.
  Questa è la tensione ai capi di R1/C5 in assenza di segnale:

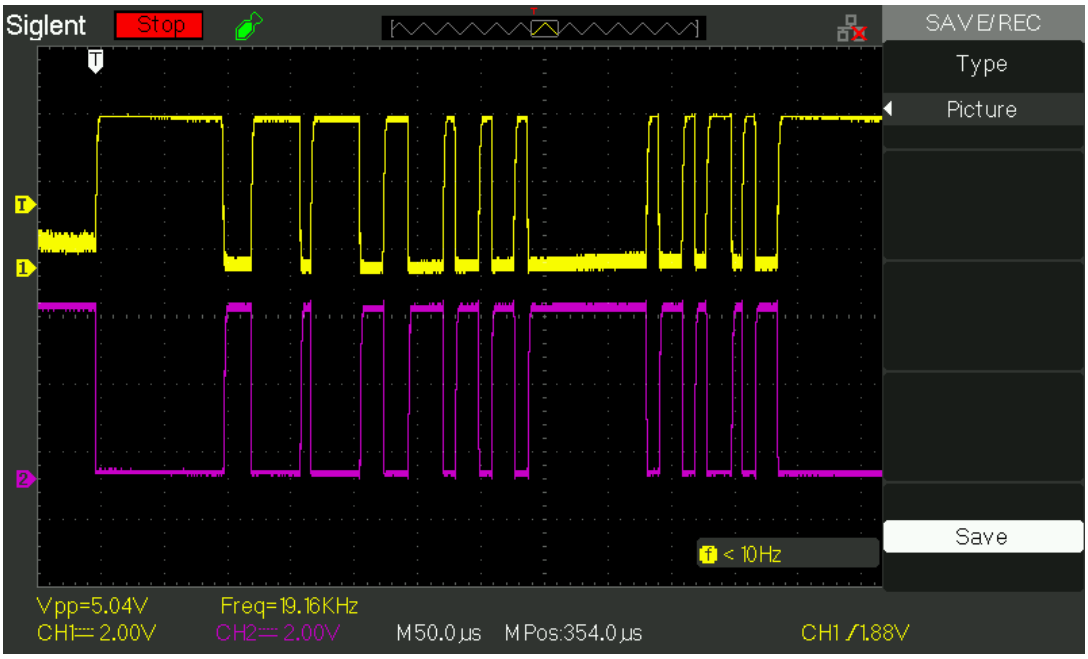


- *C7 - Rimozione dell'offset in continua*
  Il segnale in uscita al rivelatore di inviluppo presenta un offset in DC. Per questo motivo è stato utilizzato un condensatore per rimuovere questo offset.

- *R3/R4/Q1 - Amplificatore*
  La funzione del transistor Q1 è quella di amplificare l'onda quadra rilevata al livello 0V/5V richiesto
  La forma d'onda gialla è il segnale presente alla base di Q1, la forma d'onda viola è il segnale presente al collettore di Q1.



- *R5/Q2 - Invertitore di segnale*
  Il mosfet Q2 è usato come invertitore di segnale.
  La forma d'onda gialla è il segnale presente al collettore di Q1, la forma d'onda viola è il segnale presente al drain di Q2.



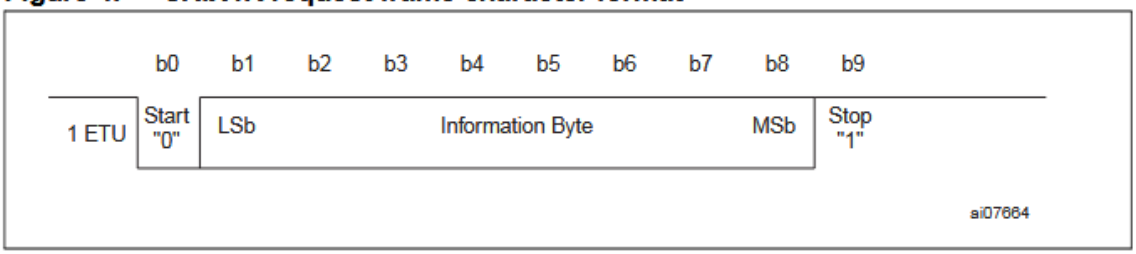
Come transistor ho usato il 2N7000, è meglio usare un mosfet di segnale con una capacità di ingresso minore.

- *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 nel gate del mosfet Q3 e "carica" il circuito dell'antenna con la resistenza R2 per generare la risposta al lettore.
- *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

Per la demodulazione del segnale viene usata la UART dell'ATMEGA.
Se notate, il frame di comunicazione fra il lettore e il tag è lo stesso di quello di una UART:

Figure 4. SRIX4K request frame character format



Ho un bit di START a zero, 8 bit di comunicazione e un bit di STOP. Il "Request start of frame" e il "Request end of frame" hanno il bit di STOP a zero:

Figure 5. Request start of frame

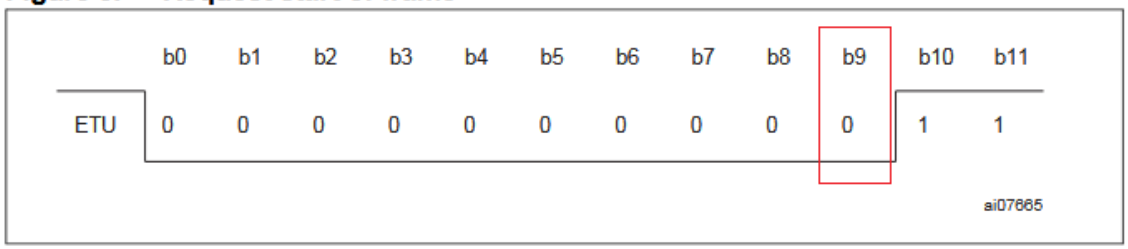
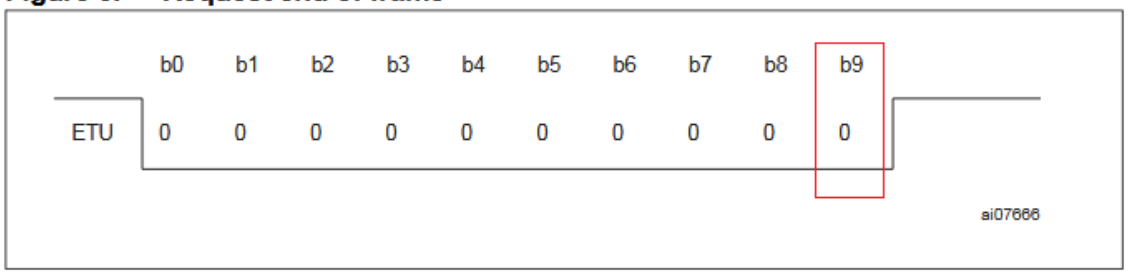


Figure 6. Request end of frame



Questa condizione di errore (bit di STOP a zero invece che a uno) viene usata per rilevare l'inizio e la fine di una comunicazione (nel registro UCSR0A della UART è presente un bit che viene messo a uno quando viene ricevuto un frame con lo STOP bit a zero).

La UART deve essere configurata per avere un BAUD rate pari a 13.56MHz/128. Il registro UBRR deve essere uguale a 7.

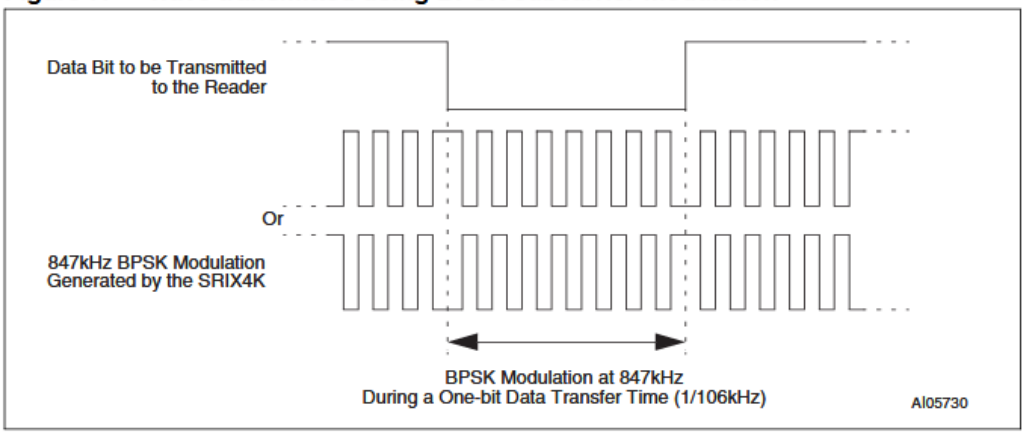
Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRRn Value
Asynchronous normal mode (U2Xn = 0)	$\text{BAUD} = \frac{f_{OSC}}{16(\text{UBRRn} + 1)}$	$\text{UBRRn} = \frac{f_{OSC}}{16\text{BAUD}} - 1$

$$UBRR = \frac{13.56MHz}{16 \times \frac{13.56MHz}{128}} - 1 = 7$$

Modulazione del segnale

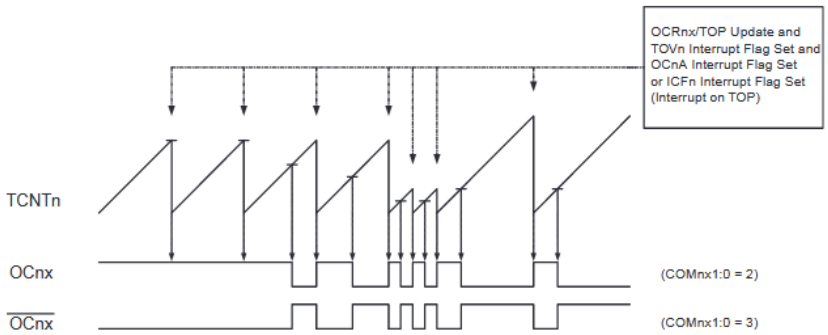
Per la generazione della risposta del tag, è necessario generare un segnale modulato BPSK, come descritto nel datasheet.

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

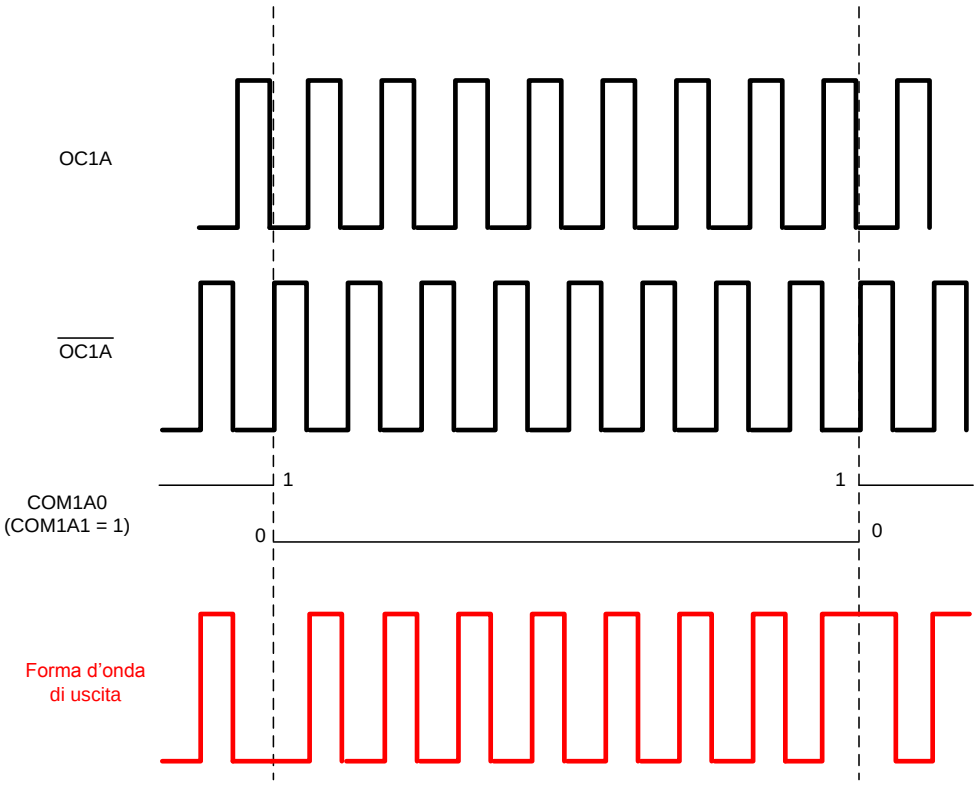


Utilizzando 15 come valore di TOP e 7 come valore di OCRA si riesce a generare un'onda quadra a (13,56 MHz)/16 = 847,5 KHz
Di per se la generazione dell'onda è semplice, il problema è la transizione di fase. Per fare questo, ho sfruttato la possibilità di invertire la fase dell'onda quadra che viene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

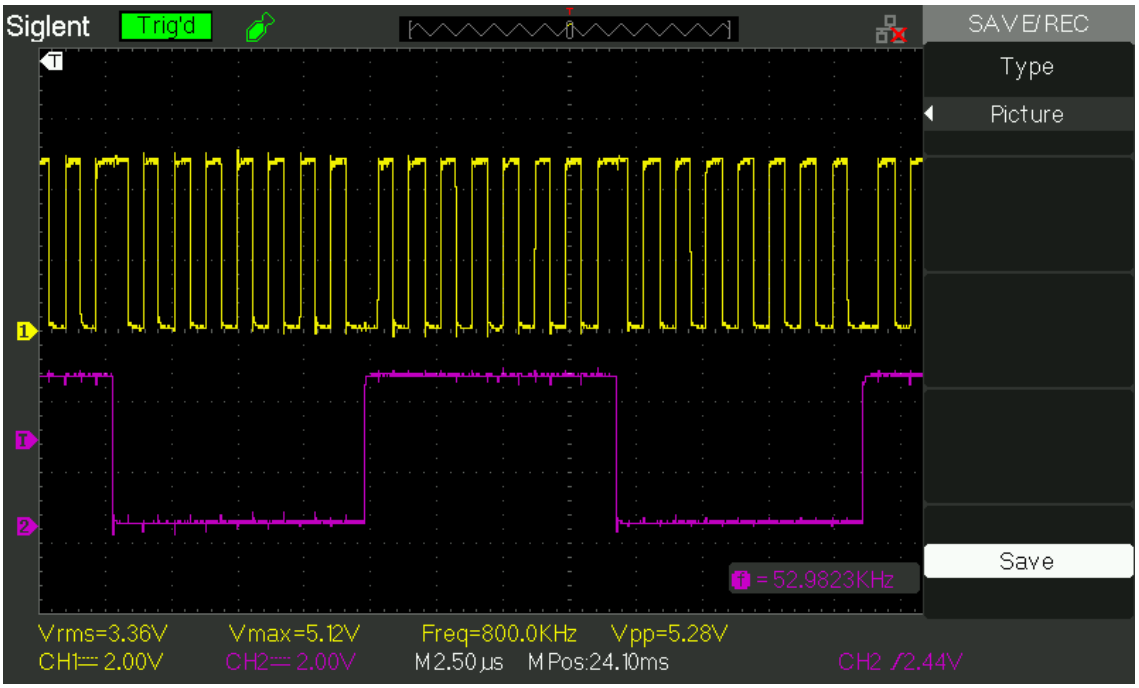
Nella routine di interrupt viene settato a uno o a zero il bit COM1A0 (è un bit del registro TCCR1A) in base allo stato della variabile **tx**, la quale cambia il suo valore (0/FF) ogni volta che l'interrupt viene eseguito. Lo stato di **tx** viene portato sul pin 0 della porta C:

```
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.

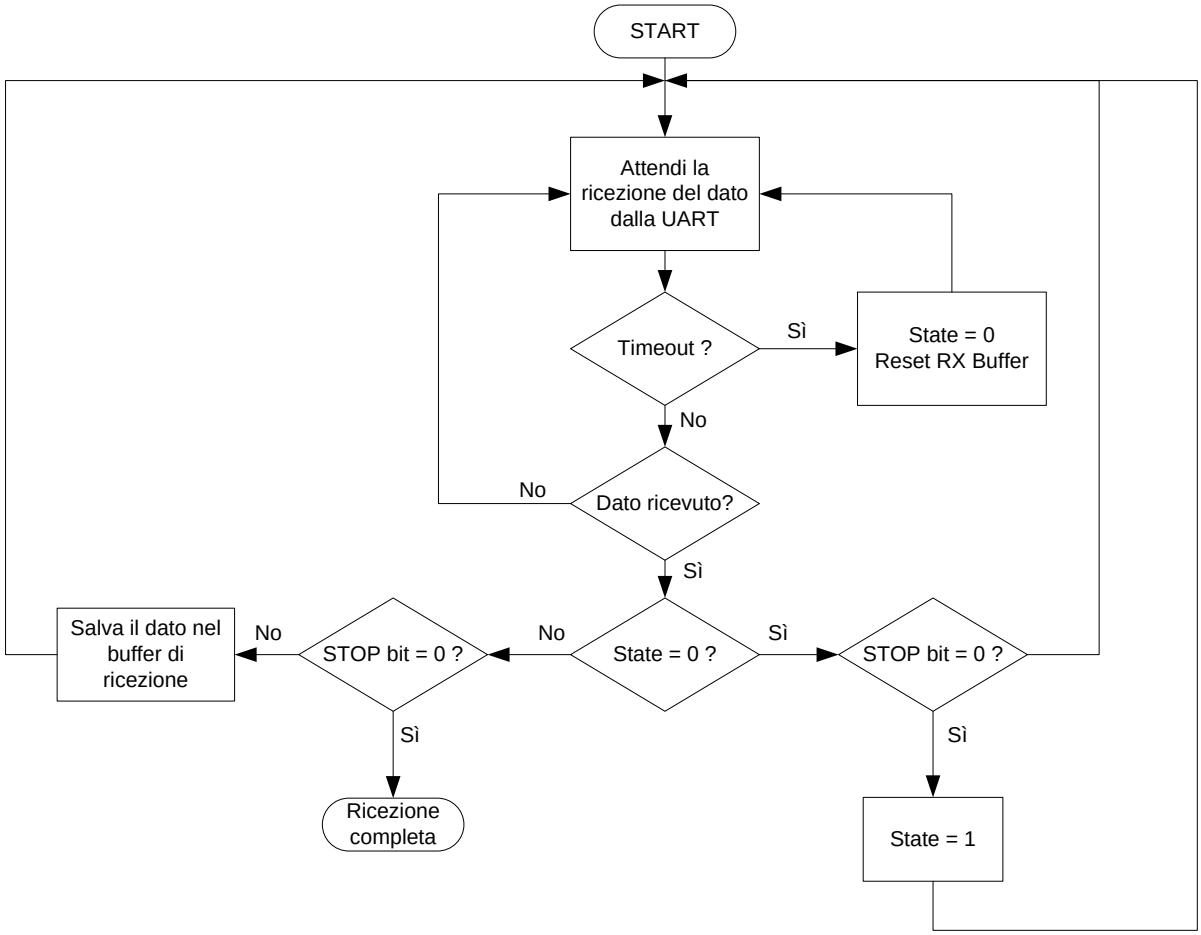
Codice finale

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, la variabile *mem_FF* utilizzata per rappresentare l'indirizzo 0xFF della memoria e la variabile UID utilizzata per rappresentare l'UID del tag.
- *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. È una seriale realizzata via software (la UART è impegnata nella ricezione del frame), la trasmissione avviene sul pin PB0. Se volete utilizzarla, utilizzate la funzione UART_init() prima di ogni trasmissione (questa seriale utilizza il TIMER0 per cui fate attenzione).
  Il codice è stato preso da qui: https://github.com/MarcelMG/AVR8_BitBang_UART_TX
  Il BAUD rate è 9600.
- *main.c*
  Programma principale.

In figura è mostrato il diagramma di flusso usato per la ricezione dei dati dalla UART:



Inizialmente il micro rimane in attesa della ricezione del dato da parte della UART. Prima di iniziare la ricezione, viene configurato il TIMER0 con un divisore del clock pari a 64 e il contatore viene inizializzato in maniera tale da generare un overflow dopo circa 140us.

Questo contatore viene utilizzato come "timeout" per filtrare eventuali disturbi sulla linea UART: un frame di comunicazione dura circa 100us, se l'attesa per la ricezione di un frame è maggiore di questo tempo (in realtà aspetto circa 140us, lascio un po' di margine) viene "resettata" la logica di ricezione e l'indice del buffer in cui vengono salvati i dati in ingresso.

```
while (1)
{
    setup_timer0();
    //Wait data from serial port
    while (!(UCSR0A & (1<<RXC0))) {
        if (TIFR0 & 0x01) //Check timer overflow
        {
            state = 0;
            num_rx = 0;
            error = 1;
            break;
        }
    }
}
```

Per la logica di ricezione vengono usati due "stati": nello stato 0 non faccio nulla e rimango in attesa dello StartOfFrame (frame con STOP bit = 0), nello stato 1 salvo i dati nel buffer di ricezione (rx_buffer) e, se rilevo un EndOfFrame (frame con STOP bit = 0), finisco la ricezione e passo alla decodifica del comando ricevuto.

```
status = UCSR0A;
data = UDR0;
if (status & ((1<<FE0))) //Stop bit = 0
    invalid = 1;

//Stop timer0
TCCR0B = 0;

switch (state)
{
    case(0):  //Wait SOF
        if (invalid && !error)
            state = 1;
        else
            state = 0;
    break;
    case(1):  //RX DATA
        if (invalid && !error) {
            EnOF = 1; //End of communication
            state = 0;
        }
        else {
            //Save data
            rx_buffer[num_rx] = data;
            num_rx++;
        }
    break;
}
```

Una volta che i dati vengono ricevuti, 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] = (CHIP_ID  + 256) << 1;
data_tx[4] = (112 + 256) << 1;
data_tx[5] = (241 + 256) << 1;
data_tx[6] = 0;
data_tx[7] = 7;

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 (nel caso in cui CHIP_ID è uguale a 0x11):

```
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] = 0000000111;
```

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 1110000000

- 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] = 7;
```

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

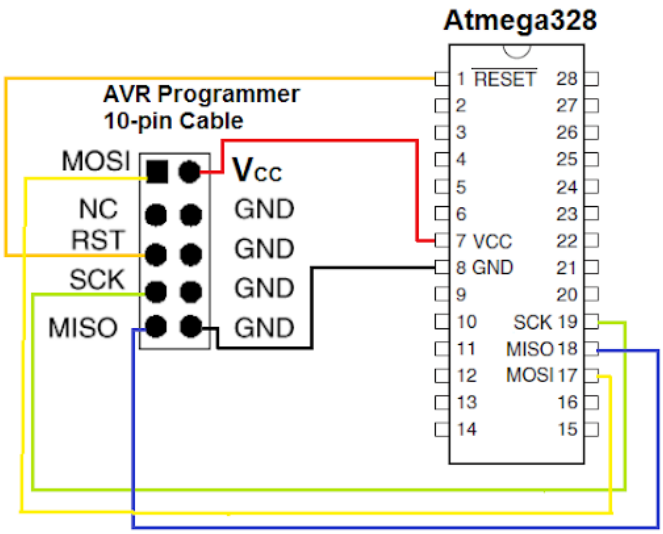
- 17 (0x11) = Chip ID
- 112 (0x70) = CRCL
- 241 (0xF1) = CRCH

Miglioramenti possibili

- Ricavare VCC direttamente dall'antenna senza usare un'alimentazione esterna.
- Aggiungere il controllo del CRC in ricezione.
- 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.