Realizzazione di un Telecomando IR che utilizza il protocollo NEC

Mattia Valeri

30/09/2023

1. Obiettivo

“Realizzare un telecomando che trasmetta codici NEC in base al tasto digitato su un tastierino 4x3. Il keypad deve essere gestito in background e non usare funzioni di libreria ed evitando rimbalzi. Le funzioni di trasmissione su led dovranno essere sincronizzate a un clock ed essere quindi indipendenti e temporizzate tramite Interrupt del clock. “loop()” dovrà essere quindi nulla.”

2. Analisi del problema

2.1 Analisi preliminare ed individuazione problemi

Per affrontare al meglio il progetto, ho deciso di dividerlo in diversi problemi più piccoli, più facili da trattare. Ne ho individuati principalmente 4: la lettura del Keypad, la gestione del Rimbalzo, la generazione della Frequenza Portante ed infine la codifica del Protocollo NEC. Le prossime sezioni saranno quindi dedicate a mostrare il mio processo di analisi rispetto questi problemi, e le mie diverse iterazioni di soluzione per ognuno fino ad arrivare all’assemblaggio del Telecomando.

2.2 Lettura del Keypad

2.2.1 Come funzionano i Keypad

Il "keypad" è un componente hardware utilizzato per l'input di dati attraverso una serie di tasti o pulsanti disposti in una matrice. I tasti possono variare in base al design e alla disposizione, ma sono generalmente organizzati in un certo numero di righe e colonne. Internamente ogni singolo tasto è collegato sia ad una riga che ad una colonna: quando un tasto viene premuto, viene creato un collegamento elettrico tra la riga e la colonna corrispondente. Se per esempio viene premuto il tasto “6” la colonna 3 e la riga 2 saranno collegate.

Figura 1: Keypad 4x3

Possiamo quindi identificare quale tasto è stato premuto dall’utente determinando quale riga e quale colonna sono collegate.

2.2.2 Scan del Keypad

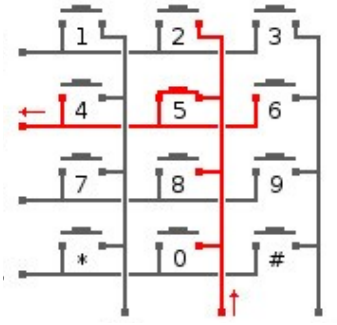


Figura 2: Circuito interno semplificato di un Keypad

Leggere un keypad coinvolge un processo di scansione delle righe e delle colonne. Scansionandole alternativamente, si può individuare quale pulsante è stato premuto. Prima ancora di iniziare il processo di scansione si configurano i pin delle righe come OUTPUT in stato HIGH e i pin delle colonne come INPUT con resistenze PULL-UP collegate. Fatto ciò la funzione di lettura porterà la prima riga a LOW procedendo a leggere lo stato delle colonne; se viene trovata una colonna in LOW allora l’incrocio delle due individua il tasto che è stato premuto, sennò si va avanti portando la prossima riga a LOW e ripetendo il processo. La funzione restituirà il numero del tasto premuto (se trovato): se viene cliccato il prima tasto verrà restituito 0, se viene cliccato il quinto verrà restituito 4. Se nessun bottone viene premuto, verrà restituito un valore che indica lo stato in cui nessun tasto viene cliccato.

2.2.3 Prima versione

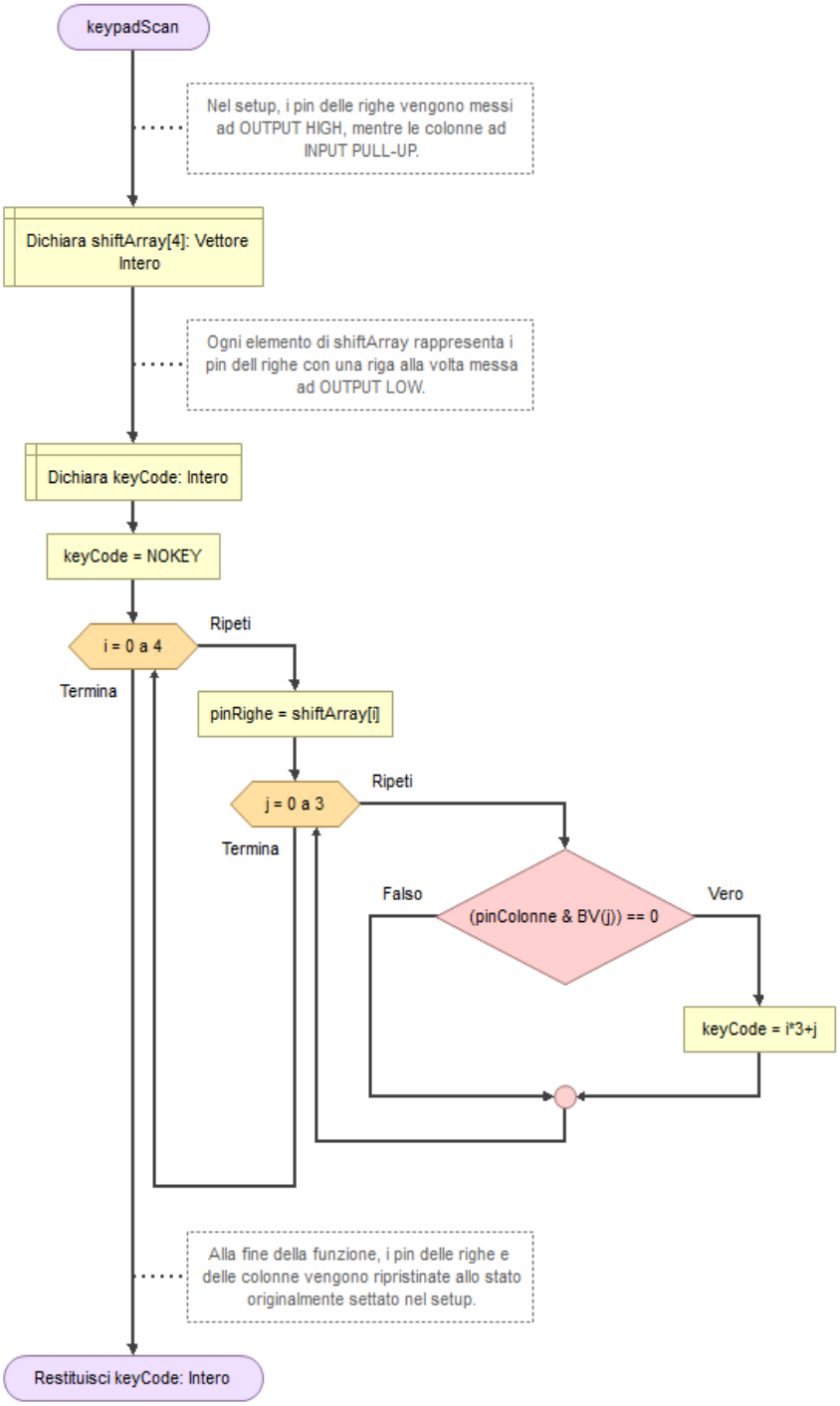


Figura 3: KeypadScan v1

Questa è la prima versione della funzione “keypadScan()”, che ha il compito di leggere il keypad e restituire il numero del tasto premuto (nel nostro caso con un keypad 3x4, il valore restituito sarà sempre $0 \leq x < 12$). Il flowchart è stato costruito riferendosi al processo spiegato in Sezione 2.2.2. Dichiariamo shiftArray e keyCode, che inizializziamo rispettivamente con i pin delle righe con una riga alla volta settata a LOW, e con NOKEY, valore che indica lo stato in cui nessun tasto è premuto (nel nostro caso può essere un qualunque valore ≥ 12). Si procede entrando in un ciclo, e settando i pin delle righe al primo elemento di shiftArray (e quindi mettendo la prima riga a LOW) e si va avanti entrando in un secondo ciclo dove controlleremo se una delle colonne è a sua volta a LOW; se la troviamo verrà calcolato il numero del tasto premuto tramite la formula “*i*3+j*”, sennò si va avanti con i cicli. La formula è scritta così perché, i valori in una riga sono 3, di conseguenza dobbiamo moltiplicare il valore della riga a LOW (quindi di i) per 3, infine per individuare il bottone, aggiungiamo al valore di prima, il valore della colonna a LOW (quindi di j).

Dal flowchart sono andato avanti scrivendo il codice: https://wokwi.com/projects/373237778887434241

Il codice rispecchia quasi perfettamente le operazioni progettate in precedenza nel flowchart. Tutti i passaggi sono commentati. Da notare che, per il corretto funzionamento, i pin delle colonne devono essere collegati al “contrario”. Questo di fatto non è un problema, ed è in ogni caso correggibile modificando _BV(j) nella condizione all’interno del secondo ciclo, in _BV(2-j).

2.2.4 Seconda Versione

Questa è la seconda versione della funzione “keypadScan()”. Durante la revisione del flowchart e basandomi sul codice che ho scritto per la prima versione ho notato che lo shiftArray contiene 4 elementi dove c’è uno 0 che semplicemente cambia, verso destra, di una posizione alla volta; per questa operazione non ho bisogno di tenere in memoria i valori, ma posso ricavarli direttamente con una semplice operazione bitwise: “*NOT(_BV(6-i))*”: l’obiettivo di tale operazione è quella di riprodurre ciò che lo shiftArray teneva in memoria, shiftando 1 di 6-i posti, quindi creando un valore che avrà un 1 sul pin della riga che dobbiamo cambiare, e 0 su tutti le altre posizioni. Questo è esattamente il contrario dei valori che avevo in precedenza in shiftArray, di conseguenza non mi basta che invertirli tutti con un NOT, per ottenere esattamente ciò che cercavo.

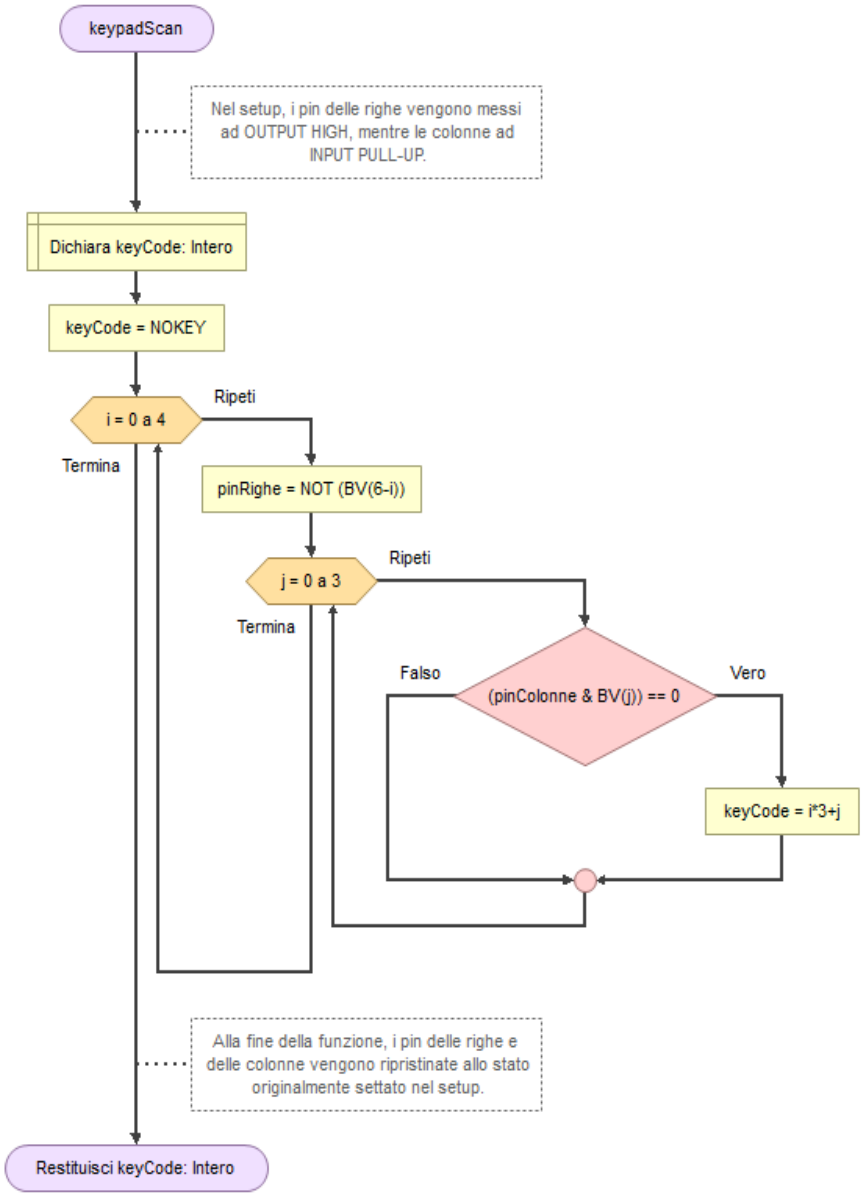


Figura 4: KeypadScan v2

Ideata questa modifica, l'ho applicata anche al codice: https://wokwi.com/projects/373257162349205505

Una ulteriore modifica di cui tenere nota è che nella prima versione, a fine funzione, ripristinavo DDRD e PORTD agli stati originariamente settati in setup, ma DDRD non viene mai cambiato in primo luogo, e di conseguenza, si può omettere l’istruzione che lo ripristina.

2.3 Gestione dei rimbalzi

2.3.1 Il fenomeno del “rimbalzo”

Un problema comune che hanno pulsanti, relè e più in generale ogni tipo di interruttore meccanico è quello del “rimbalzo”: quando un dispositivo elettromeccanico viene aperto o chiuso, i suoi punti di contatto sono soggetti a slancio e forze elastiche che, agendo insieme, creano rapide oscillazioni, facendoli quindi rimbalzare una o più volte prima di stabilire un contatto stabile. Nonostante questa oscillazione di norma duri solamente pochi millisecondi, essa genera una serie di veloci impulsi che fluttuano fra voltaggio alto e basso. Se contassimo ognuna di queste transizioni come una pressione di un bottone, otterremmo un input errato ed è quindi d’obbligo “filtrare” questi impulsi per il corretto funzionamento del bottone. Per ogni pulsante la quantità e le tempistiche di rimbalzo è diverso, e non è quindi un problema che può essere precisamente generalizzato ma esistono comunque diverse soluzioni per risolvere la questione.

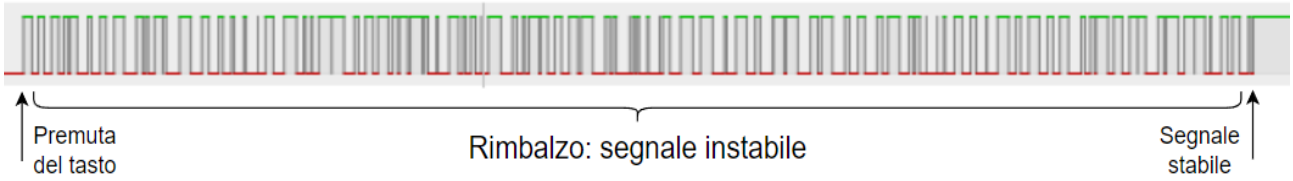


Figura 5: Esempio di rimbalzo dopo aver premuto un tasto

2.3.2 Soluzioni di Debouncing

Esistono diverse soluzioni di “Debouncing” o “Anti-Rimbalzo” create per fare in modo che un singolo premuto o rilascio del pulsante non venga interpretato come una serie di segnali; le soluzioni possono essere sia Hardware che Software:

A livello hardware la soluzione è introdurre “circuiti di debouncing”. Costruiti solitamente con componenti come condensatori, resistenze e flip-flop essi introducono un “ritardo” controllato nell’elaborazione del segnale del pulsante. Questo ritardo consente al segnale di stabilizzarsi prima che venga considerato valido, eliminando così i segnali generati dal rimbalzo.

A livello software le soluzioni coinvolgono l’implementazione di “algoritmi di debounce”. Ne esistono di diverse:

1. L’implementazione più semplice è “emulare” ciò che fanno le soluzioni hardware e semplicemente introdurre un delay di x millisecondi nella lettura del pulsante, così che abbia tempo di stabilizzarsi.
2. Un’altra soluzione è una sorta di “evoluzione” della prima: appena si legge una qualunque transizione, il bottone viene considerato premuto/rilasciato, e poi si attende x millisecondi per leggere una qualsiasi altra transizione, ignorando il resto degli altri segnali generati dal rimbalzo dopo il primo.
3. Un’ennesima implementazione è quella di osservare se lo stato del segnale rimane stabile a HIGH o LOW per x millisecondi di seguito. Questa soluzione si basa sul fatto che, per la loro natura elastica, le transizioni create dal rimbalzo sono estremamente brevi, quindi non rimarranno per molto tempo stabili su uno dei due stati; se osserviamo che il segnale è rimasto stabile per più tempo di seguito, allora il rimbalzo è considerato “terminato”.

Tutti i casi software hanno in comune il bisogno di “contare” x millisecondi per avere successo nel loro ruolo; ma quanto tempo dovrebbe effettivamente essere questi “x millisecondi”? Questo può dipendere da dove viene utilizzato il tasto in questione: per esempio il delay che può essere utilizzato sui tasti di una tastiera sarà sicuramente minore rispetto quello di un telecomando. Da tenere in considerazione è anche la quantità di ritardo percepibile dall’occhio umano che nella gran parte dei casi, non riesce a registrare delay minori di 50ms. Non esiste quindi una “formula” fissa nella scelta di questo ritardo, e dipende molto da una serie di fattori.

2.3.3 Prima Versione

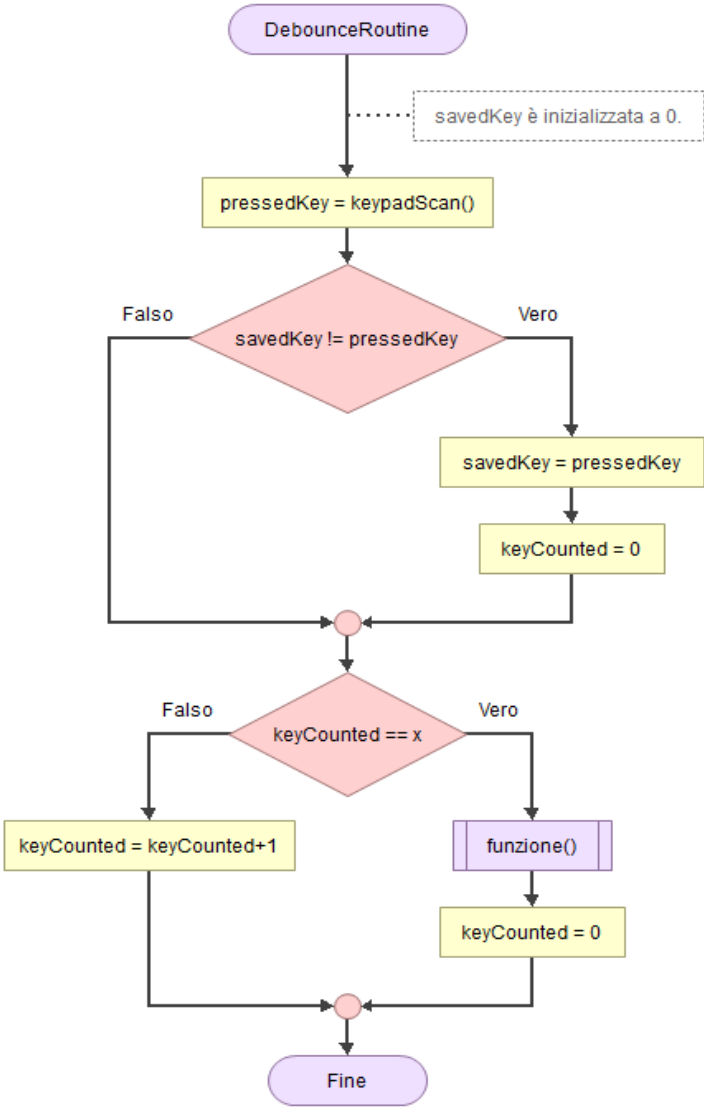


Figura 6: Debouncing v1

Avendo a che fare con un keypad che ha molteplici righe di tasti, le soluzioni software sono sicuramente più semplici e meno costose delle soluzioni hardware, che introdurrebbero molteplici circuiti identici per ogni riga/colonna. Ho deciso di utilizzare la soluzione n°3; A sinistra il flowchart costruito basandosi sul implementazione in questione. Come primo passo eseguiamo la funzione keypadScan, (analizzata nella Sezione 2.2) che ci restituirà il tasto premuto, dopo di che controlliamo se il tasto premuto equivale al tasto salvato in precedenza: se non è così, il conteggio delle volte di seguito che il tasto è rimasto uguale viene resettato, e salviamo il nuovo valore, se invece è così passiamo a controllare il numero di volte che il tasto è rimasto uguale: se questo numero equivale ad una costante x allora possiamo eseguire la nostra funzione sicuri di aver “eliminato” il rimbalzo, sennò aumentiamo di 1 il contatore. Questa prima “bozza” di implementazione è leggermente diversa da quella descritta nella Sezione 2.3.2, infatti invece di analizzare lo stato del segnale, analizziamo quale tasto è stato premuto, e se quest’ultimo rimane premuto per un certo periodo di tempo. Questa è una modifica necessaria considerato il funzionamento del keypad, e della sua funzione di Scan.

Questa routine è progettata per essere eseguita in background ogni x millisecondi. Per implementare questo tipo di soluzione avremo bisogno di un Timer che possiamo impostare per eseguire periodicamente del codice; l’ ATmega328P ha a sua disposizione 3 Timer distinti: Timer0 e Timer2 da 8bit e Timer1 da 16. La scelta di uno rispetto l’altro dipende dal lavoro che si deve soddisfare; in questo caso, non abbiamo bisogno di grande precisione o di tempi di Timer lunghi, quindi ci basta uno dei due Timer ad 8bit; utilizzerò quindi il Timer0.

Basandomi sull'analisi precedente, ho scritto il codice: https://wokwi.com/projects/370153166290765825

Nel codice imposto il Timer in modalità CTC (Clear Timer on Compare Match). Ho scelto questa modalità perché essa mi consente di scegliere un valore di confronto che, una volta raggiunto dal timer, effettua la routine di Interrupt, permettendomi quindi di poter selezionare nel modo più preciso possibile l'intervallo fra le routine. Per fare in modo che venga eseguita una routine di Interrupt ad ogni raggiungimento del valore di confronto col Timer0 bisogna cambiare 2 bit del registro TIMSK0:

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 7: Registro TIMSK0

I bit OCIE0A ed OCIE0B attivano rispettivamente Interrupt on Compare Match A e B. Nel codice ho cambiato il bit OCIE0A quindi attivando l'interrupt A. Ora è da decidere ogni quanto eseguire la routine di Interrupt. Per fare ciò dobbiamo impostare il prescaler e il valore di confronto. Il prescaler è un "divisore di frequenza" che riduce la velocità del clock del timer rispetto alla velocità del clock del microcontrollore di x volte. E' necessario scegliere il valore di prescaler giusto in modo da ottenere la frequenza di clock desiderata. In questo caso ho impostato il prescaler a 64, quindi il valore di timer verrà aumentato ogni 64 cicli di clock. Per cambiare il prescaler nel Timer0 è necessario cambiare 3 bit del registro TCCR0B:

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Figura 8: Schema bit necessari per il prescaler del registro TCCR0B

Per ottenere un prescaler di 64, non basta che cambiare i bit CS00 e CS01 ad 1. Infine, il valore di confronto nel Timer0 (per Interrupt A) viene contenuto nel registro OCR0A. Il valore di questo registro può essere calcolato dalla seguente formula per ottenere il tempo desiderato: $\mathrm{OCR0A} = \frac{frequenzaCPU}{Prescaler \cdot Frequenza\,Timer} - 1$. Ho deciso di voler eseguire la routine di interrupt ogni millisecondo, quindi con la frequenza della CPU di 16MHz, il prescaler a 64, e la Frequenza di Timer desiderata di 1kHz (ovvero $\frac{1}{1ms}$) abbiamo: $\frac{16\,000\,000\,Hz}{1 \cdot 1000\,Hz} - 1 = 249$. Infine, l'interno della routine di Interrupt segue le operazioni che avevamo progettato precedentemente col flowchart. Nel codice la costante x, usata nel secondo confronto del flowchart, è trasformata in una istruzione preprocessore #DEFINE, col valore di 5; questo significa che, con la routine ogni millisecondo, se un tasto viene osservato premuto per 5 millisecondi di seguito viene considerato senza rimbalzo, e viene permessa l'esecuzione della funzione all'interno della condizione. I valori della DEFINE e del Timer non sono vincolati da nessuna particolare ragione, e possono essere cambiati con relativa libertà. "5" ed "1ms" è solo una delle infinite combinazioni che funzionano per eliminare il rimbalzo con questa implementazione.

2.4 Generazione della Frequenza Portante

2.4.1 Frequenza Portante e Modulazione

Il telecomando a infrarossi (IR) è un dispositivo che utilizza onde infrarosse per inviare comandi seguendo un protocollo specifico. Il protocollo definisce come i segnali vengono trasmessi e ricevuti tra dispositivi. Tuttavia definire un protocollo non è abbastanza, poiché ci sono molte fonti di onde infrarosse nell'ambiente, come luci, il Sole e altre sorgenti di calore, che il ricevitore potrebbe captare al posto del segnale del telecomando; è quindi necessario garantire che il segnale trasmesso sia distinguibile dagli altri segnali. Questo viene ottenuto utilizzando una "Frequenza Portante", un segnale costante e stabile, su cui vengono modulate le informazioni da trasmettere. La frequenza portante agisce come base su cui trasportare le variazioni che rappresentano l'informazione da inviare. Questo processo di combinare il segnale di informazione con la frequenza portante è chiamato modulazione. La modulazione è importante perché permette di trasmettere il segnale di informazione attraverso il mezzo di comunicazione scelto, variando le caratteristiche della frequenza portante in risposta alle variazioni nel segnale di informazione.

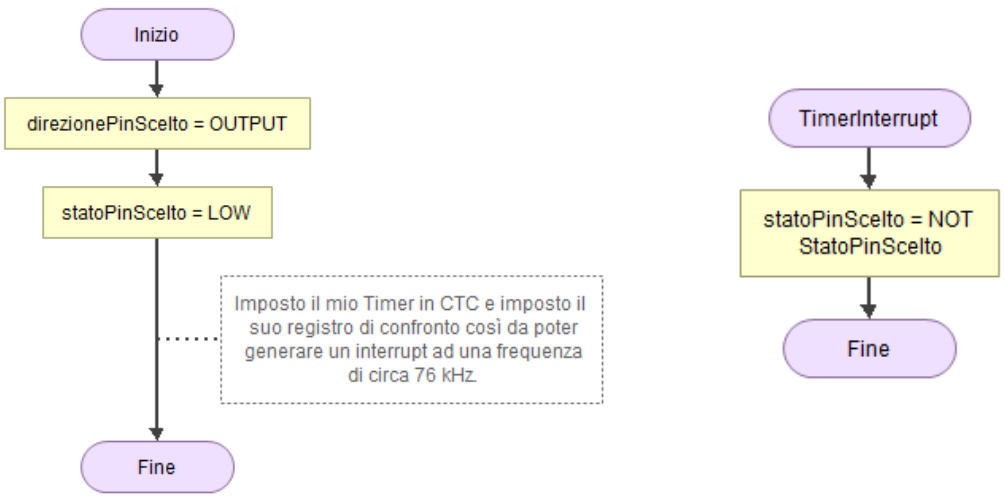
2.4.2 Modulazione del segnale secondo il protocollo NEC

Il protocollo NEC utilizza la "Pulse Distance Encoding" (Approfondito in Sezione 3.1) Ogni comando è rappresentato da una sequenza di impulsi infrarossi, dove la durata dell'impulso attivo (alto) e la durata del periodo complessivo del segnale definiscono la codifica del comando. L'impulso attivo è modulato ad una precisa frequenza consigliata di più o meno 38 kHz (la frequenza portante), così da poter distinguere il segnale infrarosso dagli altri che si trovano naturalmente nel ambiente. Quindi, ogni volta che c'è da trasmettere un impulso attivo (alto), dobbiamo in realtà trasmettere un segnale che va in stato logico alto e poi stato logico basso ad una frequenza di più o meno 38kHz. Per generare un segnale di questo genere utilizzeremo un Timer che possiamo impostare per eseguire periodicamente determinate operazioni ad una frequenza specifica.

2.4.2 Prima Versione

Per generare la frequenza portante, abbiamo bisogno di un onda quadra che vada in stato logico alto e poi stato logico basso per un periodo di circa $\frac{1}{38\,kHz} = 26{,}3\,us$, e per generare i singoli stati logici abbiamo quindi bisogno di un Timer che va al doppio della frequenza: $\frac{1}{76\,kHz} = 13{,}15\,us$.

Quel che possiamo fare per generare la frequenza portante è quindi scegliere un pin dove mandare l'output della frequenza ed impostare un Timer in modo da eseguire una routine di Interrupt ad una frequenza di 76kHz che cambia alternativamente lo stato del pin ad HIGH e LOW:



Codice scritto seguendo il ragionamento sopra: https://wokwi.com/projects/373866071670925313.
Nel codice viene utilizzato il Timer2 a 8bit che viene configurato in modalità CTC (Clear Timer on Compare Match). Ho scelto proprio il Timer2, perché il Timer0 è già utilizzato dalla routine di Debouncing e perché non ci serve la precisione del Timer1 a 16bit. Ho poi configurato il prescaler ad 1. Con un prescaler di 1 la velocità del clock non verrà ridotta. Per cambiare il prescaler nel Timer2 è necessario cambiare 3 bit del registro TCCR2B:

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk_{T2S}/(no prescaling)
0	1	0	clk_{T2S}/8 (from prescaler)
0	1	1	clk_{T2S}/32 (from prescaler)
1	0	0	clk_{T2S}/64 (from prescaler)
1	0	1	clk_{T2S}/128 (from prescaler)
1	1	0	clk_{T2S}/256 (from prescaler)
1	1	1	clk_{T2S}/1024 (from prescaler)

Figura 9: Schema bit necessari per il prescaler del registro TCCR2B

Per ottenere un prescaler di 1 non ci basta quindi che cambiare CS20 ad 1. Infine, il valore di confronto nel Timer2 viene contenuto nel registro OCR2A. Utilizzando la formula precedentemente studiata nella Sezione 2.3.3 e con la frequenza della CPU di 16MHz, il prescaler ad 1, e la Frequenza di Timer desiderata di 76kHz abbiamo: $\frac{16\,000\,000\,Hz}{1 \cdot 76000\,Hz} - 1 \approx 209$. Il periodo in realtà sarà leggermente diverso ai 38kHz desiderati (infatti con 209 in OCR2A la frequenza sarà più vicina ai 76.19 kHz e quindi un periodo di 38.095kHz circa), ma questa è un imprecisione che è accettata da virtualmente ogni tipo di ricevitore IR e non c'è quindi bisogno di preoccuparsene. All'interno della routine l'unica cosa che si fa è quella di invertire lo stato del pin scelto con un operazione XOR. Se analizziamo ora il segnale generato potremo verificare che il suo periodo è effettivamente di circa 38kHz (26,3 us):

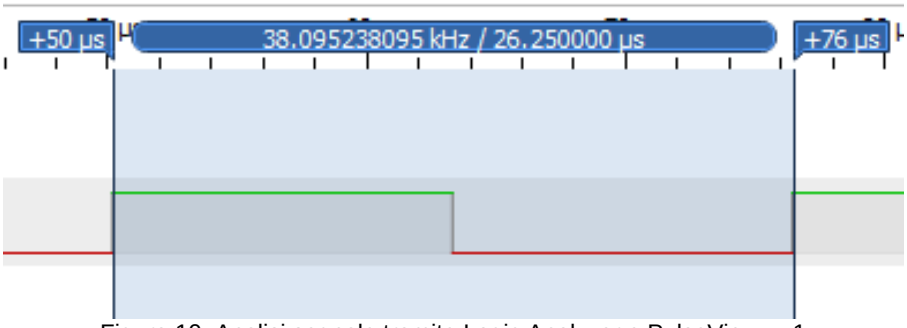


Figura 10: Analisi segnale tramite Logic Analyzer e PulseView - v1

2.4.3 Seconda Versione

Leggendo meglio il datasheet dell'ATmega328P riguardo i Timer ho scoperto che ognuno di esso ha a disposizione due pin chiamati "Output Compare Pin" (OCxA e OcxB); una volta "attivati" questi pin nei registri dei corrispettivi Timer sovrascrivono il normale funzionamento del pin a cui sono connessi eseguendo varie funzioni una volta raggiunto il valore di confronto. Nel caso del Timer2 i due registri OC2A ed OC2B sono connessi rispettivamente ai pin 11 e 3; ho scelto di utilizzare OC2A perché OC2B è già utilizzato dal keypad. Per attivare le varie funzioni di questi pin è necessario cambiare 2 bit nel registro TCCR2A:

COM2A1	COM2A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC2A on compare match
1	0	Clear OC2A on compare match
1	1	Set OC2A on compare match

Figura 11: Compare Output Mode (In modalità CTC)

Quello che cerchiamo è di alternare lo stato di OC2A ad ogni Compare Match, quindi non ci basta che cambiare COM2A0 ad 1 per attivare la modalità “Toggle OC2A on Compare Match”.

Codice scritto applicando le conoscenze citate sopra: https://wokwi.com/projects/373934213836770305
In questa nuova versione del codice non abbiamo più bisogno di eseguire una routine di interrupt ad ogni Compare Match per alternare lo stato del pin perché questa funzionalità è ora soddisfatta direttamente dal Timer, viene quindi rimossa la routine e il comando che la attiva. Analizzando ancora una volta il segnale generato possiamo verificare che il periodo rimane anche in questa versione sui 38kHz circa:

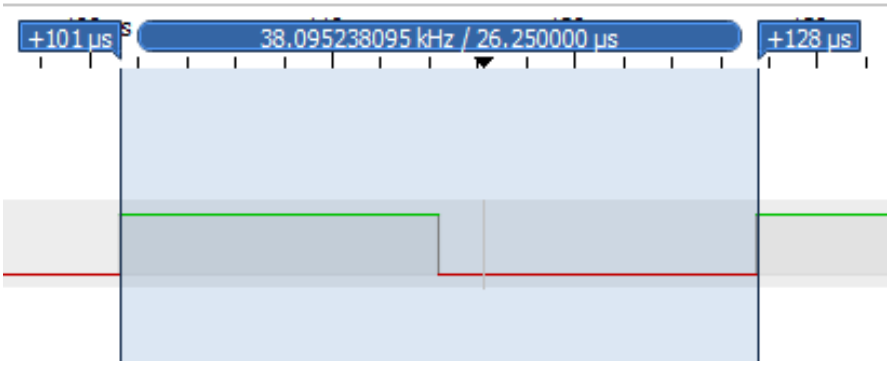


Figura 12: Analisi segnale tramite Logic Analyzer e PulseView - v2

3. Realizzazione Telecomando

3.1 Il Protocollo NEC

3.1.1 Analisi Protocollo

Il protocollo NEC è uno dei tantissimi protocolli “Consumer IR” che utilizzano la porzione degli Infrarossi dello spettro elettromagnetico per comunicazioni wireless. Il protocollo NEC in specifico è utilizzato per le comunicazioni fra telecomandi e TV. Esso codifica ogni tasto in un formato frame a 32 bit: 8 bit per l’Indirizzo, 8 bit per il complemento dell’indirizzo, 8 bit per il comando ed infine 8 bit per il complemento del comando. “Indirizzo” rappresenta un valore univoco associato ad un qualunque modello specifico di telecomando, mentre “Comando” rappresenta l’azione specifica che si desidera eseguire, come spegnere o aumentare il volume. La trasmissione dei complementi di entrambi garantisce la correttezza dei dati e sono usati per rilevare eventuali errori nella trasmissione, di fatto permettendo al ricevitore di eseguire una sorta di operazione di “checksum”. Ogni bit trasmesso è “codificato” utilizzando “Pulse Distance Encoding”: quando il LED IR è “acceso”, esso sta trasmettendo luce in impulsi dettati dalla Frequenza Portante (Analizzata nella Sezione 2.4) ed è quindi in uno stato chiamato “Impulso”; quando il LED IR è “spento” esso non sta trasmettendo luce ed è quindi in uno stato chiamato “Spazio”. Questi Spazi e Impulsi non sono tradotti direttamente in 0 e 1 perché il dato trasmesso è in realtà “codificato” secondo uno specifico metodo che determina come vengono rappresentati gli 1 e gli 0 in termini di Spazi e Impulsi.

Nel “Pulse Distance Encoding” la lunghezza dell’Impulso è sempre la stessa, ma lo Spazio fra un Impulso e l’altro è differente a seconda della trasmissione di un 1 o di uno 0. Nel caso specifico del protocollo NEC, uno “0 logico” è trasmesso come 562.5µs di Impulso seguito da 562.5µs di Spazio, mentre un “1 logico” è trasmesso come 562.5µs di Impulso e 1,6875ms di Spazio (3 volte la lunghezza di Impulso).

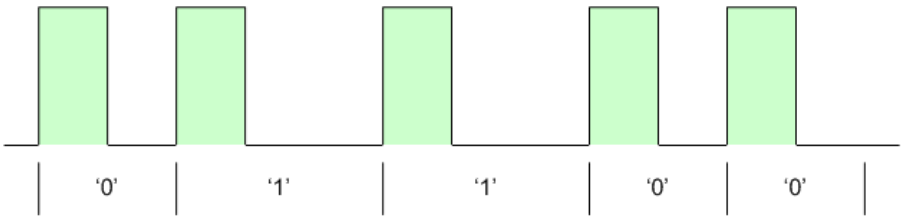


Figura 13: Esempio di codifica "Pulse Distance"

Oltre ai 32 bit di Indirizzo e Comando nel messaggio vengono trasmesse anche ulteriori “informazioni”, infatti l’intero messaggio trasmesso consiste in:

1. 9ms di Impulso Iniziali (16 volte la lunghezza di Impulso di codifica di un bit logico)
2. 4.5ms di Spazio Iniziali (8 volte la lunghezza di Impulso)
3. 8 bit di Indirizzo
4. 8 bit di complemento del Indirizzo
5. 8 bit di Comando
6. 8 bit di complemento del Comando
7. Un Impulso finale di 562.5µs che indica la fine del messaggio.

I 4 byte di dati sono trasmessi partendo dal bit meno significativo (per esempio b00000001 verrà in realtà trasmesso come b10000000). Ci vuole un totale di 67.5ms per trasmettere l’intero frame (togliendo l’Impulso finale) e 27ms per trasmettere sia i 16 bit di Indirizzo+Complemento che i 16 bit di Comando+Complemento.

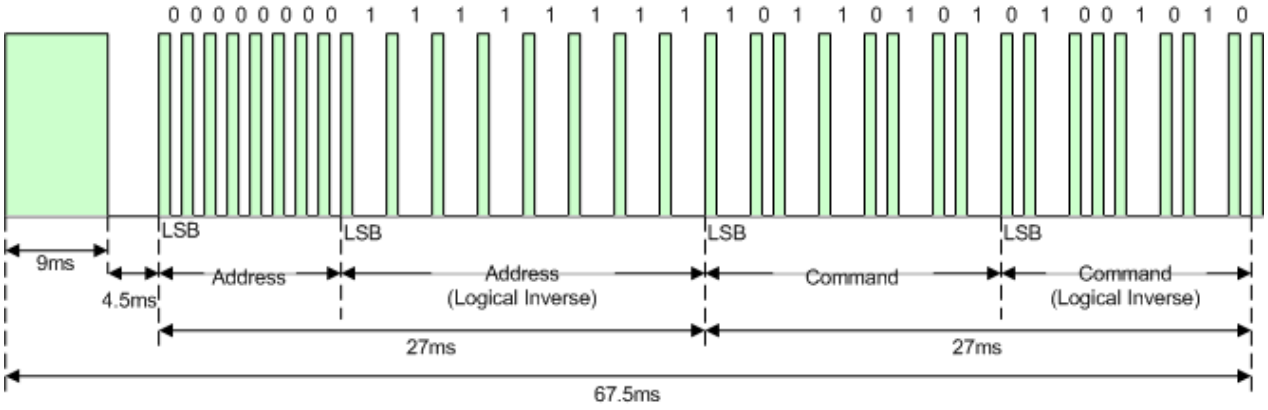


Figura 14: Esempio di un intero messaggio NEC

3.1.2 Riprodurre un messaggio NEC

Dato che ogni “segmento” di un messaggio NEC è composto o da Impulsi dove semplicemente accendiamo il LED IR in “intervalli” dettati dalla Frequenza Portante, o Spazi dove lasciamo il LED IR spento, il problema del creare un messaggio NEC si riduce essenzialmente allo spegnere e riaccendere il Generatore di Frequenza Portante (Spiegato nella Sezione 2.4.3) a precisi intervalli. Analizzando inoltre tutte le tempistiche del protocollo possiamo osservare che è tutto divisibile per la lunghezza di un singolo Impulso di codifica di un bit logico, ovvero 562.5µs:

I 9ms di Impulso iniziale non sono altro che 16*562.5µs, i 4.5ms di Spazio che succedono l’Impulso iniziale equivalgono a 8*562.5µs e anche gli 1.6875ms di uno Spazio di codifica di un “1 logico” è semplicemente 3*562.5µs. Ciò che si può quindi fare, considerate le analisi precedenti, è dividere la trasmissione in Fasi, per ogni Fase contare precisamente x volte 562.5µs per entrambi gli stati di Impulso e Spazio e, di conseguenza, accendere e spegnere il generatore di Frequenza Portante. Per esempio, partiamo accendendo il generatore, poi contiamo 16 volte 562.5µs, spegniamo il generatore e contiamo altre 8 volte 562.5µs; abbiamo riprodotto con successo la prima “fase”, ovvero, i primi due segmenti della trasmissione, 9ms di Impulso e 4.5ms di Spazio, e possiamo quindi procedere così per il resto della trasmissione.

3.2 Analisi Componenti Richiesti

Considerando che il Protocollo NEC utilizza una codifica “Pulse Distance” sappiamo con certezza che ogni Impulso sarà seguito da uno Spazio, quindi di fatto l’intera trasmissione può essere divisa in più segmenti che comprendono i due stati; da qui ci basta quindi tenere le quantità di tempo su cui il segmento rimane su entrambi gli stati (Quindi quanto rimane su stato Impulso e quanto tempo rimane su stato Spazio), e poi accendere e spegnere il Generatore di Frequenza Portante secondo queste due tempistiche. Inoltre, riprendendo ciò che abbiamo analizzato nella Sezione precedente, sappiamo che in realtà tutte queste quantità di tempo sono divisibili per 562.5µs, quindi invece di sapere l’effettiva quantità di tempo su cui rimanere su uno stato, ci basta solamente sapere quante volte è divisibile questo periodo di tempo per 562.5µs. Abbiamo quindi bisogno di una routine che ci tiene conto di questo numero e che, una volta raggiunto il numero di 562.5µs desiderati, ci spegne o accende il Generatore. Per eseguire questi compiti abbiamo bisogno di 3 specifici componenti:

1. Un componente che tiene conto del numero di 562.5µs trascorsi e che spegne/accende il Generatore di Frequenza Portante una volta raggiunto il tempo desiderato per uno Stato;
2. Un componente che imposta il numero desiderato di 562.5µs da trascorrere nell’attuale Stato secondo la Fase che abbiamo attualmente raggiunto nella trasmissione;
3. Un componente che ci tiene conto della Fase in cui siamo attualmente nella trasmissione.

Questo però basta solamente per quei segmenti che rimangono fissi per tutte le trasmissioni, quindi i due segmenti iniziali e il segmento finale, mentre il modo in cui vengono trasmessi i 4 byte di Address e Comando possono cambiare, e abbiamo quindi anche bisogno di:

4. Un componente che ci calcola il numero di 562.5µs “dinamicamente” per ogni singolo bit di questi 4byte, su ogni trasmissione.

Nelle prossime sezioni descriverò più in dettaglio ognuno di questi componenti, allegando anche il flowchart.

3.2.1 Variabili

Ogni componente avrà bisogno di tenere conto di diversi valori e tempi, e quindi ognuno utilizzerà diverse variabili. Le variabili sono counter1, counter2, l'array da due posizioni stateCounter, burstCount, bitCount ed infine phaseCount. Il lavoro di ognuna verrà esposta nelle prossime Sezioni insieme all'analisi dei componenti stessi.

3.2.2 Routine di Conteggio

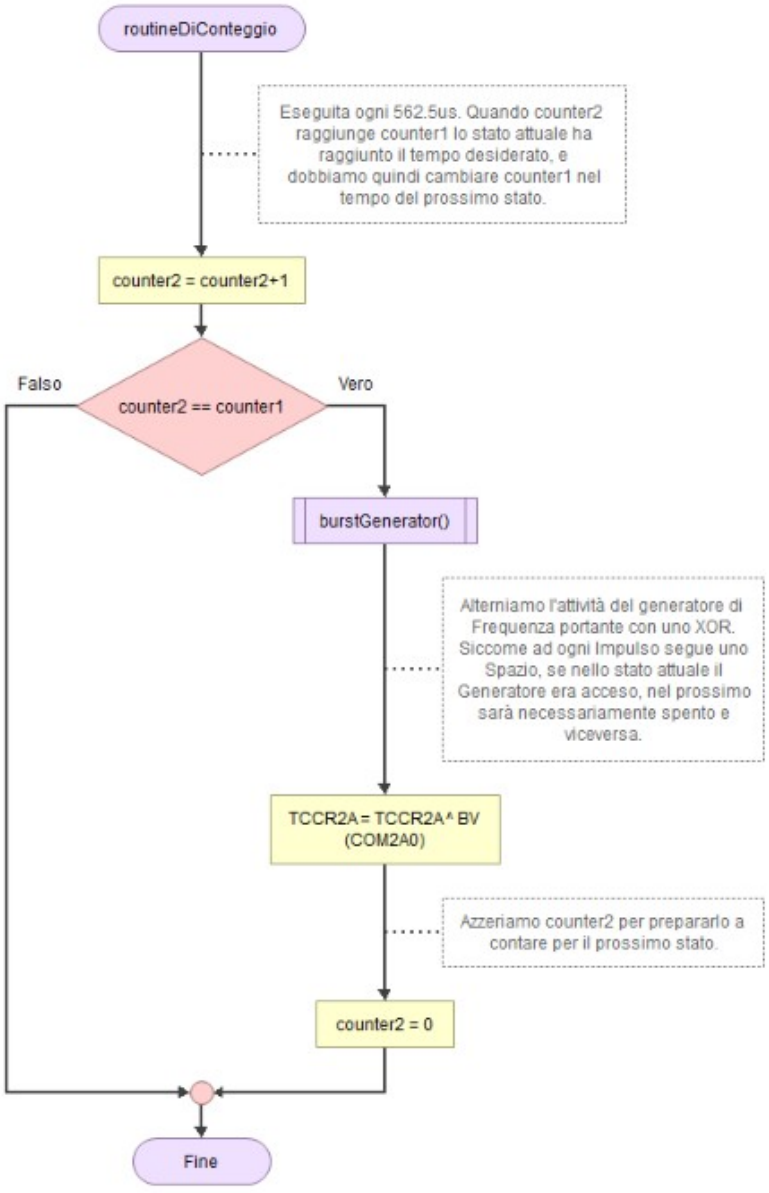


Figura 15: Routine di Conteggio

La routine di Conteggio è al cuore del nostro problema e della nostra analisi. Il suo funzionamento si basa sull'osservazione fatta nella Sezione 3.1.2 della divisibilità per 562.5µs di tutte le tempistiche del protocollo NEC. La routine viene eseguita appunto ogni 562.5µs e aumenta counter2; quando quest'ultimo raggiunge counter1 allora lo stato attuale ha raggiunto il tempo desiderato e si va avanti cambiando counter1 tramite la funzione burstGenerator (descritta nella prossima Sezione) e facendo il resto delle operazioni descritte nel flowchart.

Per esempio immaginiamo di dover mandare i 9ms di Impulso e i 4.5ms di Spazio iniziali. Iniziamo accendendo il generatore di Frequenza Portante, settiamo counter1 a 16 ($\frac{9\,ms}{562.5\,\mu s}=16$), e la routine inizia ad aumentare counter2 ogni 562.5µs; quando anche essa raggiunge 16, viene chiamato burstGenerator, che cambia counter1 nel valore del prossimo stato, cioè 8 ($\frac{4.5\,ms}{562.5\,\mu s}=8$) spegne il Generatore e azzera counter2 per prepararlo a contare per il prossimo stato e va avanti così per l'intera durata della trasmissione. Per eseguire una routine periodicamente in tempi prestabiliti possiamo utilizzare un Timer. Siccome il Timer0 è utilizzato per le routine di Anti-Rimbalzo, e il Timer2 per la generazione della Frequenza Portante, non ci resta che utilizzare il Timer1 a 16bit.

3.2.3 BurstGenerator

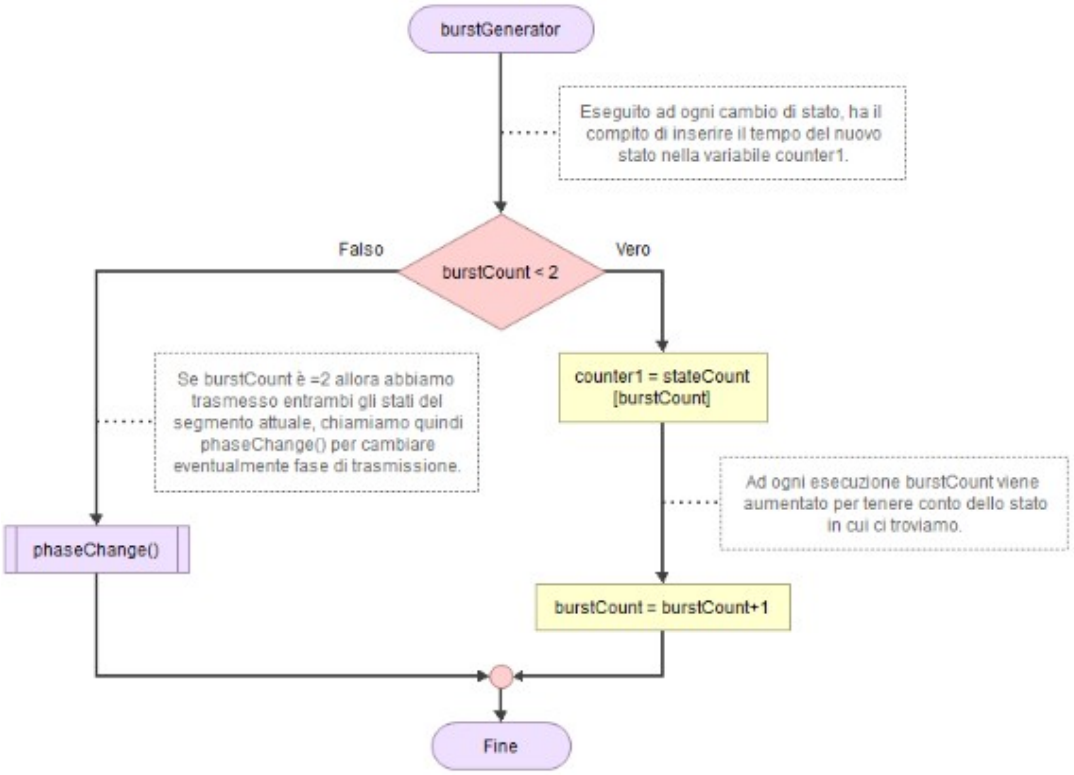


Figura 16: BurstGenerator

Il BurstGenerator ha il compito di cambiare il valore di counter1 ad ogni cambio di stato, permettendo alla routine di Conteggio di spegnere ed accendere il generatore di Frequenza Portante nelle tempistiche giuste. Esso viene eseguito ad ogni cambio di stato. Il suo funzionamento si basa sull'analisi del "Pulse Distance Encoding": ogni informazione è codificata in un periodo di Impulso ed un periodo di Spazio, ed abbiamo quindi massimo 2 stati di cui tenere conto. Il resto del funzionamento è già spiegato nel flowchart stesso.

3.2.4 buildNec

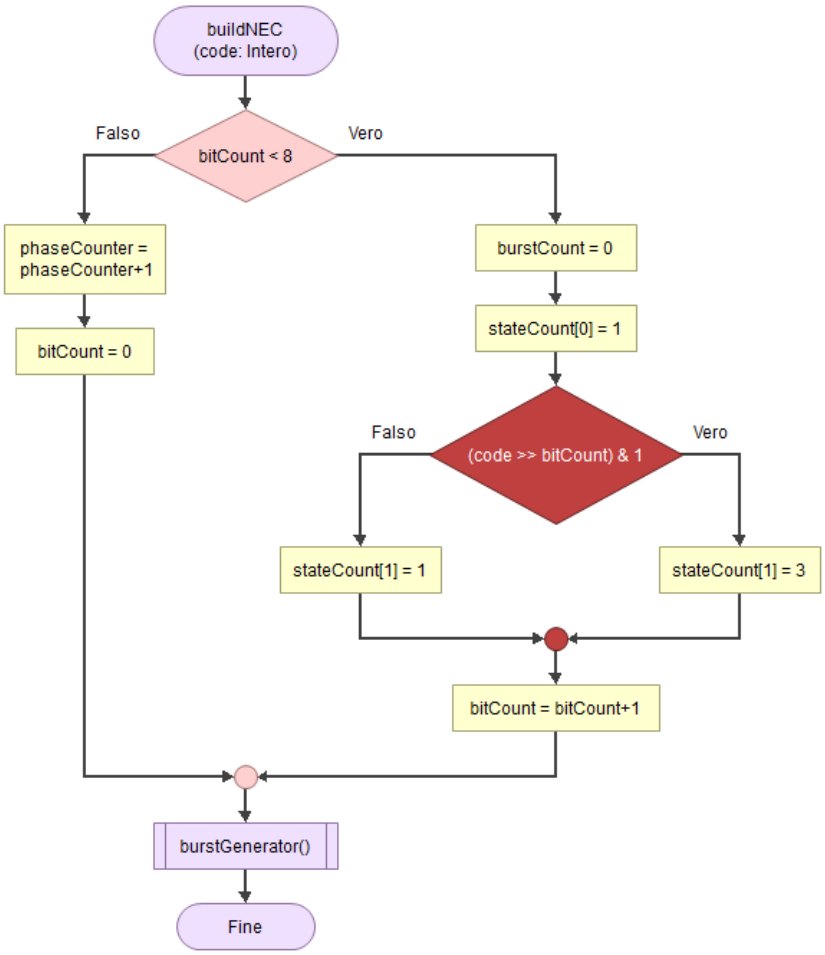


Figura 17: buildNec

BuildNEC ha il compito di codificare ogni bit dei 4byte di Indirizzo e Comando. La funzione riceve come parametro un byte alla volta. Quando ci sono ancora bit da trasmettere (bitCount<8), prima di tutto prepariamo il burstGenerator per la trasmissione dei due stati, azzerando il burstCount, poi inseriamo “1” come primo valore nell’array che contiene i due stati , questo perché nel Protocollo NEC, utilizzando la codifica “Pulse Distance”, la lunghezza del Impulso è uguale sia in un 0 logico che in un 1 logico, ed infatti equivale per entrambi a 562.5µs; dopo di che si controlla se il bit che dobbiamo attualmente codificare è un 1 o uno 0: code,che è il byte intero ricevuto come parametro, shiftato a destra di bitCount “elimina” tutti i bit già eventualmente analizzati, dandoci sempre il bit giusto, infine l’AND con 1 restituirà 1 (quindi VERO) se il bit in questione è 1 (1&1=1) FALSO se il bit è 0 (1&0=0). Se il bit è 0 allora come secondo valore nell’array inseriremo 1, sennò 3 ($\frac{1.6875\,ms}{562.5\,\mu s}=3$) seguendo così la codifica dei bit del Protocollo NEC (spiegata in Sezione 3.1.1).
Si aumenta bitCount e si esegue burstGenerator per preparare il counter1 ai valori appena inseriti in stateCount. Una volta analizzato l’intero byte (bitCount=8) aumentiamo phaseCounter andando avanti alla prossima fase della trasmissione e azzeriamo bitCount.

3.2.5 Il Conta Fasi

Come spiegato nella Sezione 3.1.2 l'intera trasmissione può essere divisa in fasi dove viene eseguita una certa operazione su ognuna di essa. Osservando l'intero messaggio di una tipica trasmissione col protocollo NEC le fasi saranno:

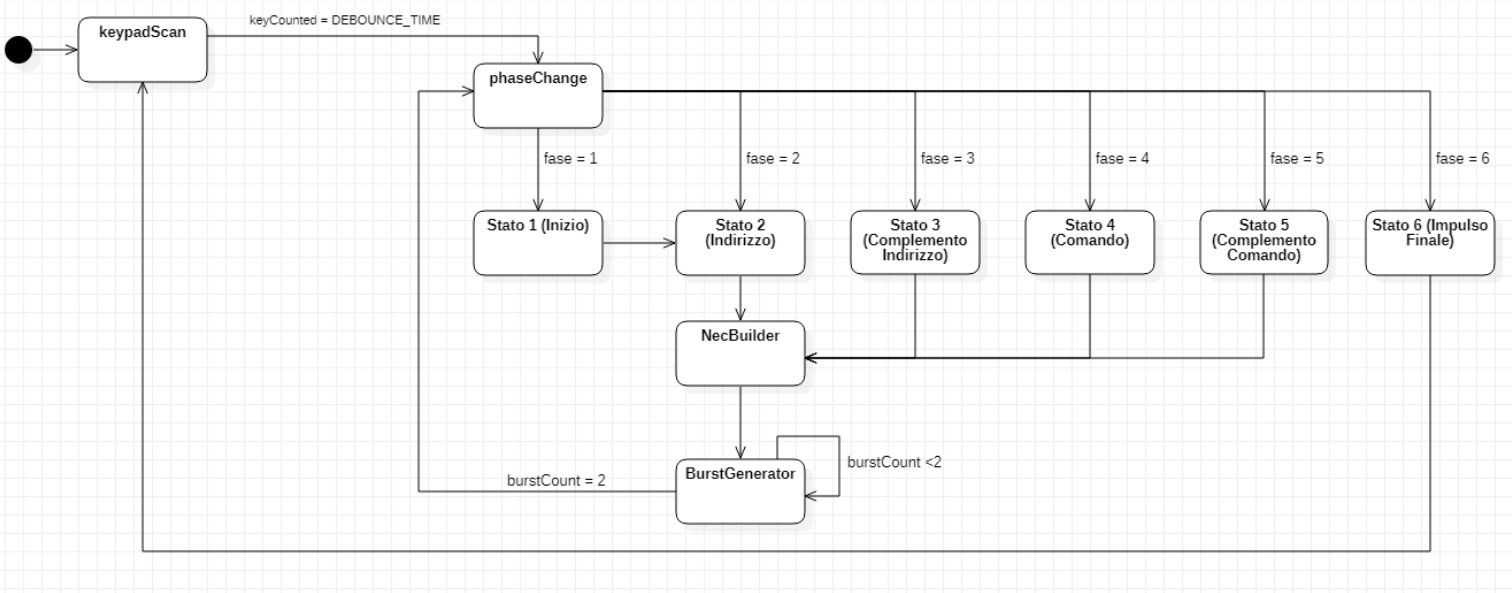
- Fase 1: 9ms di Impulso e i 4.5ms di Spazio;
- Fase 2: 8bit dell'Indirizzo;
- Fase 3: 8bit dell'Indirizzo (complemento);
- Fase 4: 8bit del Comando;
- Fase 5: 8bit del Comando (complemento);
- Fase 6: impulso di 562.5µs finale;

Abbiamo quindi bisogno di un'ulteriore funzione il cui scopo è quello di controllare la fase attuale della trasmissione ed eseguire le operazioni opportune per ognuna.

3.3 Unire tutti i Componenti

3.3.1 Statechart

Abbiamo analizzato e affrontato tutti i problemi individuati nella Sezione 2.1, progettando e costruendo i moduli che costituiscono il Telecomando. Li abbiamo creati in modo da compiere un singolo obbiettivo e per funzionare in modo indipendente; dobbiamo ora unirli e capire come farli lavorare insieme al meglio.



Questo è il statechart che rappresenta tutti gli stati che il nostro telecomando può assumere. Si esegue keypadScan e si osserva se è stato premuto un tasto, se è così se fa il debouncing, dopo di che si va al conta fasi, che esegue le varie fasi della trasmissione del messaggio NEC. Completata la trasmissione si ritorna al keypadScan per ricominciare il processo.

3.3.2 Prima Versione Codice

A questo punto ho unito tutti i componenti e ho costruito la prima versione del codice del telecomando completo: https://wokwi.com/projects/374413741439815681. Analizziamo il codice:

```
#define DEBOUNCE_TIME 5
#define NO_KEY 20
#define REMOTE_ADDRESS 0x00

const uint8_t NEC_Codes[] = {
  0x30, //1
  0x18, //2
  0x7A, //3
  0x10, //4
  0x38, //5
  0x5A, //6
  0x42, //7
  0x4A, //8
  0x52, //9
  0xA8, //Volume+
  0x68, //0
  0xE0  //Volume-
};

volatile uint8_t pressedKey = NO_KEY;
volatile uint8_t savedKey = 0;
volatile uint8_t keyCounted = 0;
volatile uint8_t key;

volatile uint8_t counter2 = 0;
uint8_t counter1 = 0;

uint8_t phaseCounter = 0;
uint8_t bitCount = 0;
uint8_t burstCount = 0;
uint8_t stateCount[2];
```

Figura 18: Preprocessore, costanti e variabili

DEBOUNCE_TIME rappresenta il tempo in millisecondi che riteniamo abbastanza per considerare l'input di tasto senza rimbalzi (Sezione 2.3);

NO_KEY rappresenta lo stato in cui nessun tasto del keypad è premuto (Sezione 2.2);

REMOTE_ADDRESS e l'array NEC_Codes rappresentano rispettivamente l'Indirizzo del dispositivo e i comandi NEC che il telecomando da a disposizione. L'indirizzo e i comandi sono separati perché l'indirizzo è uguale per tutti i comandi. I codici in esadecimale sono stati presi dal sito: https://exploreembedded.com/wiki/NEC_IR_Remote_Control_Interface_with_8051#NEC_Protocol;

La variabile pressedKey rappresenta il tasto attualmente premuto sul keypad. E' inizializzata al valore NO_KEY;

La variabile savedKey e keyCounted sono utilizzate all'interno delle routine Anti-Rimbalzo. (Sezione 2.3)

La variabile key è utilizzata per salvare il codice del comando NEC che corrisponde al tasto del keypad.

Le variabili counter1, counter2, phaseCounter, bitCount, burstCount, stateCount sono tutte utilizzate nella trasmissione del comando NEC (Sezione 3.2)

```
// Routine Anti-Rimbalzo
ISR(TIMER0_COMPA_vect) {
  pressedKey = keypad_scan();

  if (savedKey != pressedKey || keyCounted > DEBOUNCE_TIME){
    savedKey = pressedKey;
    keyCounted = 0;
  }

  if (keyCounted++ == DEBOUNCE_TIME && phaseCounter == 0 && pressedKey != NO_KEY)
  {
    key = NEC_Codes[pressedKey];
    phaseChange();
  }
}
```

```
//Funzione di scan del Keypad
uint8_t keypadScan() {
  uint8_t keyCode = NO_KEY;
  for (uint8_t i = 0; i < 4 ; ++i) {
    PORTD = ~_BV(6-i);
    for (uint8_t j = 0 ; j < 3 ; ++j) {
      if (!(PIND & _BV(j))) {
        keyCode = i * 3 + j;
        break;
      }
    }
  }
  PORTD = 0x7F;
  return keyCode;
}
```

Figura 19: Routine di Anti-Rimbalzo e keypadScan

La routine di Anti-Rimbalzo e keypadScan seguono ciò che avevamo già costruito nella Sezione 2.3, tranne per alcuni piccoli accorgimenti nella prima:

- keyCounted viene azzerata e savedKey cambiata anche se keyCounted va oltre al DEBOUNCE_TIME.
- Se phaseCounter è diverso da 0, vuol dire che c'è una trasmissione NEC in corso, e non si può cercare di iniziarne un altra.
- Per ottenere il codice del comando NEC che corrisponde al tasto premuto, lo prendiamo dall'array alla posizione pressedKey; questo è possibile perché abbiamo progettato la funzione keypadScan per ridarci il numero del tasto premuto, quindi non basta che ordinare i nostri codici nell'array secondo l'ordine dei tasti nel keypad.

```
void buildNEC(uint8_t code) {
  if (bitCount < 8) {
    burstCount = 0;
    stateCount[0] = 1;
    stateCount[1] = ((code >> bitCount++) & 1) ? 3 : 1;
  } else {
    phaseCounter++;
    bitCount = 0;
  }

  burstGenerator();
}
```

```
ISR (TIMER1_COMPA_vect) {
  if (++counter2 == counter1) {
    burstGenerator();

    TCCR2A ^= _BV (COM2A0);
    counter2 = 0;
  }
}
void burstGenerator() {
  if (burstCount < 2)  counter1 = stateCount[burstCount++];
  else phaseChange();
}
```

Figura 20: Funzioni per la trasmissione messaggi NEC

Queste sono buildNEC, burstGenerator e la routine di Conteggio tutte e tre costruite e studiate nella Sezione 3.2. Nessun cambiamento di nota rispetto la loro analisi preliminare e flowchart.

La routine di Conteggio utilizza il Timer1 da 16bit, impostato in modalità CTC e con l'Interrupt on Compare Match A attivato. Per eseguire la routine di interrupt ogni 562.5µs, dobbiamo impostare il valore del prescaler del Timer1 a 1; per fare ciò dobbiamo cambiare il valore CS10 del registro TCCR1B ad 1. Dobbiamo infine impostare il registro OCR1A correttamente. Utilizzando la formula precedentemente studiata nella Sezione 2.3.3 calcoliamo che: con la frequenza della CPU di 16MHz, il prescaler ad 1, e la Frequenza di Timer desiderata di 1.77776kHz (cioè $\frac{1}{562.5\,us}$) abbiamo $\frac{16\,000\,000\,Hz}{1 \cdot 1777.76\,Hz} - 1 = 8999$.

```
void phaseChange() {
  switch (phaseCounter) {
    case 0: { // Inizio
      stateCount[0] = 16;
      stateCount[1] = 8;
      burstGenerator();

      phaseCounter++;

      TCCR1B |= _BV (CS10);
      carrier_toggle();
    } break;

    case 1: { // Address
      buildNEC(REMOTE_ADDRESS);
    } break;

    case 2: { // ~Address
      buildNEC(~REMOTE_ADDRESS);
    } break;

    case 3: { // Codice
      buildNEC(key);
    } break;

    case 4: { // ~Codice
      buildNEC(~key);
    } break;

    case 5: { // Impulso finale
      counter1 = 1;
      phaseCounter++;
    } break;

    case 6: { // Fine
      carrier_toggle();
      TCCR1B &= ~_BV (CS10);

      counter1 = 0;
      phaseCounter = 0;
      burstCount = 0;
      bitCount = 0;
    } break;
  }
```

La funzione Conta Fasi (nominata phaseChange dopo lo stateChart) progettata in Sezione 3.2.5. Il suo scopo è tenere conto della fase in cui si trova la trasmissione, ed eseguire tutto ciò che serve per trasmettere un messaggio NEC corretto. Nella sezione 3.2.5 avevo solamente accennato a grandi linee cosa bisogna fare in ogni fase per creare un messaggio corretto, qua possiamo vederlo con più dettaglio:

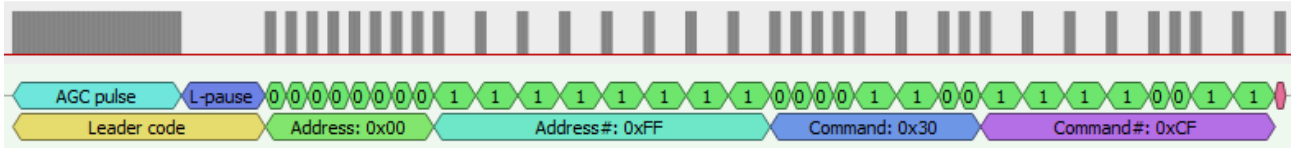
- **Fase 0:** Carichiamo in stateCount i valori 16 e 8; questi valori rappresentano i 9ms di Impulso e i 4.5ms di Spazio iniziali divisi per 562.5µs (Questo segue ciò che avevamo osservato sui tempi del protocollo NEC in Sezione 3.1.2). Eseguiamo poi burstGenerator per caricare il primo valore dello stateCount su counter1, attiviamo la routine di Conteggio ed eseguiamo carrier_toggle:

```
void carrier_toggle(){
  TCCR2B ^= _BV (CS20);
  TCCR2A ^= _BV (COM2A0);
}
```

  carrier_toggle è una semplice funzione che semplicemente attiva o disattiva completamente il generatore di Frequenza. Viene utilizzata solamente all'inizio e alla fine della trasmissione;

- **Fase 1:** Trasmettiamo l'Indirizzo;
- **Fase 2:** Trasmettiamo l'Indirizzo (Complemento);
- **Fase 3:** Trasmettiamo il Comando;
- **Fase 4:** Trasmettiamo il Comando (Complemento)
- **Fase 5:** Inseriamo in counter1 il valore 1 per permettere alla routine di Conteggio di trasmettere l'ultimo singolo impulso di 562.5µs che segna il termine della trasmissione;
- **Fase 6:** Disattiviamo completamente il generatore di Frequenza Portante, la routine di Conteggio e resettiamo tutte le variabili utilizzate al loro stato originale.

Eseguite queste 6 fasi la trasmissione è stata completata con successo. Ora che tutti i componenti sono stati uniti, possiamo testare il funzionamento del telecomando e vedere se il messaggio NEC che viene trasmesso dal LED IR è corretto:



Questo è il messaggio che il LED IR trasmette dopo aver premuto il tasto "1" del keypad. Possiamo osservare che il formato del messaggio, Indirizzo (0x00), Comando (0x30) e i loro complementari (rispettivamente 0xFF e 0xCF) sono tutti corretti.

3.3.2 Seconda Versione Codice: La ripetizione di Comando

Nel protocollo NEC, se un tasto del telecomando viene lasciato premuto dopo l'invio della prima trasmissione, un "codice di ripetizione" viene emesso, che segnalerà al ricevitore di ripetere il comando precedente. Questo codice viene emesso solitamente circa 40ms dopo l'impulso che indica la fine del messaggio principale. Il codice di ripetizione è composto da:

- 9ms di Impulso Iniziali
- 2.25ms di Spazio
- Un Impulso finale di 562.5µs che indica la fine del codice di ripetizione.

Possiamo osservare che, ritornando al concetto analizzato nella Sezione 3.1.2 sulla divisibilità di tutti i segmenti di una trasmissione NEC per 562.5µs, anche questi ultimi seguono la "regola": infatti $\frac{40\,ms}{562.5\,\mu s}=71$ e $\frac{2.25\,ms}{562.5\,\mu s}=4$; questo è un bene, perché significa che abbiamo bisogno di modificare solamente la funzione phaseChange, a cui dovremo aggiungere delle nuove fasi per poter ospitare questa nuova funzionalità.

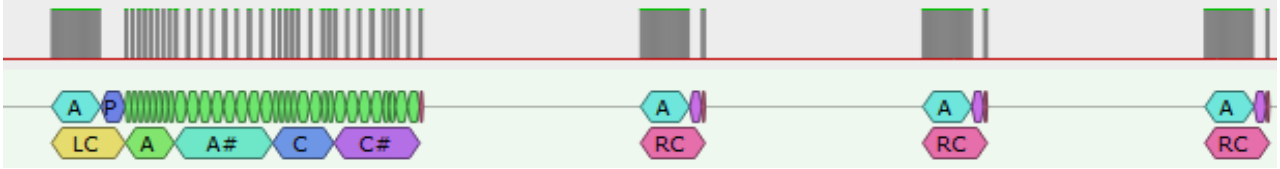
Codice Completo con ripetizione di codice: https://wokwi.com/projects/374504307076985857

```
case 5: { // Impulso finale
  counter1 = 1;
  phaseCounter++;
} break;

case 6: {
  if (NEC_Codes[pressedKey] == key) {
    counter1 = 71;
    phaseCounter++;
  } else {
    phaseCounter = 8;
  }
} break;

case 7: {
  burstCount = 0;
  stateCount[0] = 16;
  stateCount[1] = 4;
  burstGenerator();

  phaseCounter = 5;
} break;

case 8: { // Fine
  carrier_toggle();
  TCCR1B &= ~_BV (CS10);

  counter1 = 0;
  phaseCounter = 0;
  burstCount = 0;
  bitCount = 0;
} break;
```

Sono state quindi aggiunte due nuove fasi dopo la fase 5: Fase 6 e Fase 7. La fase finale, cioè la Fase 6 nella prima versione, è stata spostata alla Fase 8 nella nuova versione. Analizziamo le nuove fasi in dettaglio:

- **Fase 5:** Uguale alla prima versione.
- **Fase 6:** Controlliamo se il tasto precedentemente premuto è ancora attualmente premuto. Se è così inseriamo in counter1 il valore 71 per permettere alla routine di Conteggio di trasmettere i 40ms di Spazio, e quindi iniziare la trasmissione del Codice di Ripetizione, sennò si va alla Fase 8 per terminare la trasmissione;
- **Fase 7:** Come in Fase 0, carichiamo in stateCount i valori 16 e 4; questi valori rappresentano i 9ms di Impulso e i 2.25ms di Spazio che compongo il Codice di ripetizione. Eseguiamo poi burstGenerator e si ritorna alla Fase 5, per trasmettere un Impulso finale di 562.5us che segna il termine del Codice;
- **Fase 8:** Se il tasto non è più premuto, disattiviamo il generatore di Frequenza Portante, la routine di Conteggio e resettiamo tutte le variabili utilizzate al loro stato originale.

Ora che anche quest'ultima funzionalità è stata aggiunta testiamo la trasmissione del messaggi NEC con le ripetizioni insieme alle ripetizioni:



Messaggio che il LED IR trasmette dopo aver tenuto premuto per tempo prolungato il tasto "1" del keypad. Come si può osservare sono presenti i Codici di Ripetizione che volevamo trasmettere.