Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Controllo remoto dei condizionatori d'aria attraverso LIRC

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Ion Ursachi

Sessione II
Anno Accademico 2017-18

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Al giorno d'oggi, molti dispositivi possono essere controllati attraverso dei telecomandi che fanno uso della radiazione infrarossa per la trasmissione wireless, a breve distanza, dei dati. Questo tipo di comunicazione richiede bassa potenza, basso costo e non soffre di disturbi elettromagnetici perciò è assai conveniente. Il meccanismo di funzionamento di una trasmissione a infrarossi è abbastanza semplice; un fascio luminoso, a frequenza 38 KHz (per i telecomandi tipici) viene proiettato contro un ricevente. Questo fascio luminoso non è altro che una sequenza di bit (0 e 1) che viene decodificata dal ricevente, attraverso la polarizzazione di photo-transistor. Per esempio un codice attribuito ad un pulsante potrebbe essere: **010101001110**. Il ricevente decodifica questo codice ed esegue il comando attribuito ad esso. Tuttavia è assai inconveniente avere un telecomando per ogni dispositivo. Basta ricordare tutte quelle volte che non si trova il telecomando della TV, dello stereo, del condizionatore d'aria, ecc. Questo è uno dei motivi principali per cui è stato sviluppato LIRC. Esso offre la possibilità di avere un punto centrale per controllare la maggior parte dei dispositivi dotati di un ricevitore a infrarossi.

1.1 Scopo del progetto

L'obiettivo di questo progetto è quello di estendere e modificare LIRC in modo che riesca a riprodurre anche quei telecomandi che inviano i segnali in base ad un proprio stato interno. Quindi, dopo le nuove implementazioni, questo strumento sarà in grado di:

1. decodificare tale telecomando
2. mappare i vari pulsanti in un file di configurazione
3. creare il segnale scegliendo quale stato bisognerebbe inviare
4. inviare il segnale
5. mostrare l'ultimo stato inviato

1.2 Protocolli

Per essere riconosciuti dal ricevitore, vengono usati diversi protocolli per la codifica del segnale. Esistono 3 protocolli usati dalla maggior parte dei produttori: NEC, RC5 e RC6. In questo progetto, nella fase di decodifica abbiamo fatto riferimento solo al primo protocollo e a tutti gli altri simili ad esso, ma l'implementazione è flessibile perciò si possono aggiungere anche gli altri. La parte d'invio invece, è indipendente dal protocollo di codifica.

Capitolo 2

LIRC

2.1 Che cos'è LIRC

LIRC(Linux Infrared Remote Control) è uno strumento open source che consente di decodificare e inviare segnali a infrarossi di molti (ma non tutti) telecomandi di uso comune. La parte più importante di LIRC è il daemon lircd che decodifica i segnali a infrarossi ricevuti dai driver del dispositivo di input fornendo questa informazione su un socket ed accetta dei comandi per inviare dei segnali a infrarossi se l'hardware lo supporta. L'uso più popolare di LIRC è attraverso il Raspberry Pi per la creazione di progetti di domotica.

2.2 Telecomandi Supportati

Esistono molti file di configurazione nella base di dati di LIRC e dovrebbero coprire più di 2500 dispositivi. Questi file sono stati creati da altri utenti che hanno deciso di rendere pubblico il proprio lavoro. Tuttavia c'è la possibilità di non trovare il dispositivo desiderato nella base di dati precedente, ma questo non vuol dire che non è supportato ma semplicemente che non esiste ancora un file di configurazione per esso. Quindi bisogna attraversare tutto il procedimento di decodifica usando determinati comandi. L'operazione in sé non è molto difficile, ma richiede un po' di pazienza.

2.3 Come usare LIRC

Per poter usare questa libreria, innanzitutto bisogna avere un sistema Linux ed un apposito hardware in grado di inviare e ricevere segnali a infrarossi. Ad esempio un Raspberry Pi collegato ad un breadboard come nell'immagine seguente.

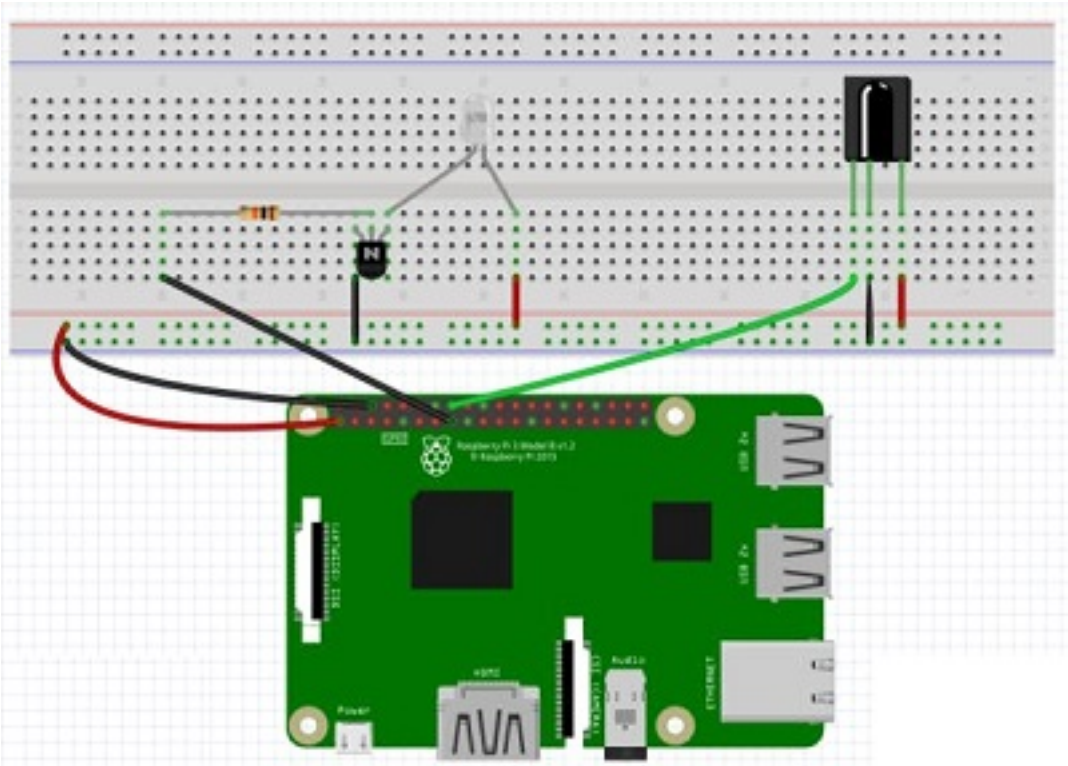

Figura 2.1: Esempio circuito Raspberry Pi

A questo punto basta seguire pochi passaggi:

1. Installare e configurare LIRC come indicato nel sito ufficiale www.lirc.org;
2. Cercare se già presente nella base di dati un file di configurazione per il proprio telecomando. Questo procedimento può essere fatto manualmente andando sul sito ufficiale oppure tramite il comando irdb-get. Nel caso in cui non dovesse esistere, questo file può essere creato tramite il comando irrecord

3. Avviare il daemon lircd

4. Infine, iniziare ad usare i comandi desiderati. Ad esempio:

   - irsend, uno strumento che invia i segnali.
   - mode2, riceve i segnali e gli mostra sullo schermo.

2.4 File di Configurazione

Il file ha due scopi: fornire informazioni sul tempo al driver in modo che possa decodificare i dati durante la pressione dei pulsanti e fornire una mappatura dai pulsanti ai simboli usati da lircd.

La struttura di base è un elenco di blocchi che individuano ciascun telecomando. Gli spazi bianchi e le interruzioni di riga vengono utilizzati come delimitatori. Ogni blocco inizia con **begin remote** e termina con **end remote**. La parte iniziale all'interno di questo blocco descrive le proprietà di base del telecomando. Di seguito è presente una sezione di codici delimitata da **begin codes** e **end codes**. Ogni riga all'interno di questa sezione contiene il nome del pulsante seguito da un valore esadecimale che corrisponde all'informazione inviata da esso.

Se il carattere di commento (#) appare come il primo carattere su una riga, il resto della riga viene ignorato.

2.5 Limiti

LIRC, pur essendo uno strumento molto potente, dà la possibilità di inviare solo segnali statici. Questo vuol dire che è in grado di riprodurre solo telecomandi che non salvano il proprio stato (i.e. TV, stereo, ecc), e cioè che un pulsante genera sempre lo stesso segnale.

Se invece, si vuole riprodurre un telecomando di un condizionatore d'aria, il quale invia un segnale in base alle informazioni salvate al suo interno

(solitamente quelle che si vedono sullo schermo del telecomando), le cose diventano più difficili. È necessario creare un comando per ogni suo stato. Ad esempio, per potere impostare la modalità "COOL" con "23" gradi, bisogna creare un codice per questa **esatta** combinazione. Per poter cambiare temperatura è necessario creare altri n codici dove si cambia solo la temperatura lasciando invariata qualsiasi altra impostazione. Questo ha come conseguenza un file di configurazione contenente centinaia di codici con informazioni ridondanti, in quanto ogni combinazione del telecomando dev'essere salvata.

Ma allora perché dobbiamo usare LIRC ? Non possiamo creare un piccolo script che spegne e accende i pin? La risposta è no. Affinché gli infrarossi funzionino correttamente, è necessario generare un segnale a 38 kHz piuttosto costante. Ciò significa che bisogna eseguire un intero loop ogni 1/38000 secondi. E che problema c'è, giusto ? Persino il primo Raspberry Pi aveva una CPU da 700 MHz che è più che abbastanza veloce da accendere e spegnere un pin a 38 kHz. Peccato però che non tutti questi 700 MHz sono per noi. È necessario eseguire il sistema operativo, che richiede una notevole quantità di velocità di elaborazione. Quindi uno script in python, o persino un programma compilato in c o c++, potrebbe essere ritardato/interrotto dal sistema operativo poiché non è l'unica cosa in esecuzione.

LIRC come fa? Utilizza un "modulo del kernel" che lavora allo stesso livello del sistema operativo. In altre parole, gira direttamente sulla CPU e può fare tutto ciò che il sistema operativo può fare. In questo modo, può impedire di essere interrotto dal sistema operativo garantendo un segnale coerente sui pin GPIO del Raspberry Pi.

Capitolo 3

Supporto telecomandi stateful per LIRC

Nel capitolo successivo spiegherò come in questo lavoro è stato esteso LIRC in modo da introdurre il supporto dei telecomandi stateful, più precisamente quelli di un condizionatore d'aria. Il progetto può essere scaricato dal seguente url: `https://github.com/nunutu29/LIRC-STATEFUL` .

3.1 Definizioni

Prima di partire introduciamo alcune definizioni per capire meglio il funzionamento delle nuove implementazioni.

Un **impulso ad infrarossi**, è formato da tanti **pulse** e **space**. Un pulse rappresenta il tempo per quanto è rimasto acceso il led. Uno space invece, è il tempo trascorso tra un pulse ed un'altro. Servono entrambe le informazioni per rappresentare un codice binario. Un pulse ed uno space, con determinati valori rappresentano un "1" oppure uno "0". Un segnale è diviso in 3 parti:

1. Uno header, che non è altro che un pulse ed uno space con valori molto più elevati degli altri presenti all'interno del segnale. Serve per farsi individuare dal proprio ricevitore.

2. Un flusso di altri pulse e space che rappresentano le informazioni da inviare al ricevente.

3. Infine un pulse finale(trailing pulse), che serve per chiudere lo space precedente, perché altrimenti che space sarebbe senza un pulse ?

Un telecomando usa il **protocollo stateless** quando non salva nessuna informazione al suo interno, ovvero un pulsante produce sempre lo stesso output. Invece usa il **protocollo stateful** quando ogni segnale è relativo al proprio stato interno del telecomando. Solitamente, ogni pulsante cambia una determinata porzione del segnale da inviare, ma invia sempre "tutto". Chiariamo meglio questo concetto con un'immagine.

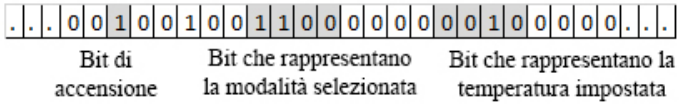

Figura 3.1: Una porzione di un codice binario di un condizionatore d'aria

3.2 Panoramica

Le nuove funzionalità sono state implementate cercando di mantenere lo standard di LIRC, cioè avere circa lo stesso file di configurazione e mantenere l'esecuzione dei nuovi comandi simile a quelli già esistenti. LIRC, come già detto in precedenza, invia un segnale per ogni codice presente nel file di configurazione, perciò per non doverlo cambiare è stato necessario creare un modo alternativo per la sua interpretazione. Questa metodologia è stata adottata per facilitarne la comprensione agli utenti che usano già LIRC.

Prima di partire è necessario cambiare alcune impostazioni. Aprire il file "/etc/lirc/lirc_options.conf" e modificare le seguenti regole:

```
driver = default
device = /dev/lirc0
```

3.3 Le nuove funzionalità

Le nuove funzioni consistono nella creazione del file di configurazione e nella sua elaborazione. Esse sono:

- Nuovo tool *stateful_create*
- 2 nuovi comandi per il tool *irsend*
    1. *stateful_send*
    2. *stateful_get*

Vediamoli ora più nel dettaglio.

3.3.1 STATEFUL_CREATE

Questo tool crea un file di configurazione denominato lircd.conf, il quale verrà letto da LIRC al momento del suo avvio. Il comando per avviare questo tool è il seguente:

```
stateful_create [options] <directory>
```

Parametri

La **directory** deve contenere alcuni file senza i quali non si può andare avanti.

1. Nel caso in cui fosse in possesso di un file **lircd.conf** valido, ottenuto tramite questo procedimento, allora basta copiare questo all'interno della directory e andare direttamente al punto 3.
2. Nel caso invece in cui, non possiede un file di configurazione, è necessario creare un file **main.sa** attraverso il tool **mode2 -m** contenente più segnali possibili.

   ```
   mode2 --driver default --device /dev/lirc0 -m > main.sa
   ```

Una volta avviato il comando sopra, puntare il telecomando verso il ricevitore a infrarossi e iniziare a premere, non troppo veloce, più pulsanti possibili. Questo file verrà usato in una elaborazione iniziale per calcolare alcune variabili importanti.

3. Serve inoltre, **un file per ogni pulsante** che si desidera salvare. Ad esempio un file **power.sa** che rappresenta il pulsante ON/OFF, oppure un file **mode.sa** che rappresenta le possibili modalità del condizionatore, ecc. Questi file si ottengono esattamente come il main.sa con la differenza che non bisogna cambiare nessun'altra impostazione quando si registra un singolo pulsante.

Oltre al file lircd.conf verranno elaborati solo i file **.sa**.

Le **opzioni** che si possono assegnare sono 5:

1. `-n <nome del dispotivo>.`

   Questa stringa è fondamentale per un funzionamento più intuitivo. Rappresenta un identificativo che verrà usato al momento dell'invio di un segnale.

2. `-h`

   Esso non ha nessun argomento richiesto, ma serve semplicemente per ottenere un messaggio di *help* contenente tutte le informazioni riguardanti questa funzione.

   `-d`

   Come l'help, nemmeno questo parametro richiede un'argomento, ma serve per indicare che si vuole ottenere un file contenente i segnali nella loro rappresentazione binaria. Impostato questo parametro nessun file di configurazione verrà creato.

3. `-s <intero>`

   Questo parametro ha come argomento un numero. Verrà usato per capire quando termina un segnale e quando inizia un'altro. Non sarà usato direttamente questo numero, ma verrà sommato al pulse del header. Di default è impostato a 5000, e la maggior parte delle volte non si deve nemmeno cambiare.

4. `-b <intero>`

   Questa è un'informazione cruciale. Serve per capire quale coppia di pulse e space rappresenta un "1" e quale uno "0". Il controllo avviene confrontando questo valore alla differenza tra il pulse e lo space. Se esso è superiore alla differenza, allora è uno "0", altrimenti è un "1". Di default è impostata a 400, tuttavia molte volte può non essere corretto. Per individuarlo è necessario aprire un file ottenuto attraverso il comando **mode2** e guardare all'incirca la differenza tra i pulse ed gli space. Prendiamo come esempio il seguente segnale:

1	3159	1550
2	535	1050
3	533	1052
4	533	288
5	535	288
6	534	288
7	535	1050
8	533	289
9	533	289
10	535	1051

11	533	1051
12	534	288
13	534	1051
14	533	289
15	535	288
16	534	1050
17	534	1052
18	534	288
19	535	1051
20	533	1051

21	536	288
22	535	288
23	534	1052
24	535	288
25	534	290
26	534	1051
27	534	289
28	535	288
29	535	287
...	...	...

Tabella 3.1: Esempio segnale a infrarossi

Questi numeri rappresentano i tempi espressi in μs. I valori nella prima colonna(esclusa quella dell'enumerazione) rappresentano i pulse mentre gli altri gli space. I primi 2 valori(quelli della riga numero 1) sono molto

più alti rispetto agli altri, quindi questi sono i valori del header, mentre gli altri appartengono al flusso di dati che si sta inviando. Notiamo subito che alcune coppie sono del tipo [∼535, ∼1050] altre invece del tipo [∼535, ∼290]. Le coppie con valori più alti rappresentano un "1", quelle con valori più bassi invece rappresentano uno "0". Facendo la differenza tra i valori di ciascuna coppia, otteniamo che la differenza nella prima è 515, mentre nella seconda è 245. Perciò il nostro valore può essere circa 400.

5. `-c <nome dell'algoritmo per il byte di controllo>`

   Tutti i telecomandi che usano il protocollo stateful, aggiungono un byte di controllo alla fine del segnale, cosi colui che deve riceverlo riesce a verificare l'integrità dell'informazione ricevuta. Purtroppo il modo di calcolarlo è quasi sempre diverso da condizionatore a condizionatore. Perciò è compito dell'utente capire l'algoritmo. Può essere utile usare il file con la rappresentazione binaria dei segnali (ottenuto usando il parametro -d) insieme ad un foglio di calcolo per ottenerlo. Ci sono alcuni algoritmi già implementati che lavorano sui numeri binari.

   - SUM_RIGHT: Somma tutti i byte, e prende gli ultimi 8 bit.
   - SUM_LEFT: Somma tutti i byte, e prende i primi 8 bit.
   - XOR: Esegue l'xor di tutti i byte, cioè prende il primo bit di ogni byte ed esegue l'xor e lo mette in prima posizione del byte di controllo, poi prende il secondo bit e fa di nuovo l'xor e lo mette in seconda posizione, ecc.
   - COUNT1_RIGHT: Conta tutti gli "1", trasforma questo numero in binario e prende gli ultimi 8 bit.
   - COUNT1_LEFT: Analogamente prende i primi 8 bit.
   - COUNT0_RIGHT: Come COUNT1, solo che conta lo "0".
   - COUNT0_LEFT: Come COUNT1, solo che conta lo "0".

- FLIP_SUM_RIGHT_FLIP: Scambia "1" con "0"(01000000 → 10111111), esegue SUM_RIGHT, e poi scambia di nuovo "1" con "0".
- FLIP_SUM_LEFT_FLIP: Analogo a quello sopra.
- REVERSE_SUM_RIGHT_REVERSE: Inverte tutti i byte (01000000 → 00000010), esegue SUM_RIGHT, e poi inverte il risultato.
- REVERSE_SUM_LEFT_REVERSE: Analogo a quello sopra.
- FLIP_REVERSE_SUM_RIGHT_FLIP_REVERSE: Scambia "1" con "0", inverte i byte, esegue SUM_RIGHT, scambia di nuovo "1" con "0" ed infine inverte il risultato.
- FLIP_REVERSE_SUM_LEFT_FLIP_REVERSE: Analogo a quello sopra.

Nel caso in cui il proprio algoritmo non dovesse risultare nell'elenco, è possibile aggiungerlo. Per farlo però, è necessario modificare il codice sorgente. Questo lo vedremo nella sezione 3.3.4

Funzionamento

Cerchiamo ora di spiegare come funziona **stateful_create**. Prima di tutto esegue il parsing dei parametri in input, prendendo dei valori di default per quelli mancanti, e legge tutti i file validi all'interno della directory. Inoltre ci prepariamo un oggetto di tipo **struct ir_remote** (stateful_create.cpp:28) che conterrà tutte le informazioni da salvare nel file di configurazione. Per comodità chiamiamolo **REMOTE**. A questo punto controlla subito se la lista contiene un file **lircd.conf**(stateful_create.cpp:896). In caso affermativo, apre il file e chiama la funzione **read_config**(stateful_create.cpp:661) la quale restituisce un puntatore ad una memoria assegnata dinamicamente di tipo **struct ir_remote** oppure NULL in caso di errori. Questa funzione esisteva già nel pacchetto LIRC, l'unica modifica fatta è stata quella di aggiungere la possibilità di leggere dal file di configurazione il nome dell'algoritmo da usare per il calcolo del byte di controllo(config_file.c:462). Se la lettura è

andata a buon fine, i dati acquisiti vengono salvati all'interno di REMOTE. Nel caso in cui il file di configurazione fosse inesistente, viene cercato un file **main.sa**. Se non lo trova, l'esecuzione termina subito con un messaggio d'errore, altrimenti inizia la fase di decodifica dei segnali. La funzione che fa questo si chiama **process_main**(stateful_create.cpp:910) e restituisce vero o falso in base al proprio esito. Prima di tutto calcola la lunghezza dei segnali e il valore che gli separa(vedi parametro -s), dopodiché crea una lista contenente tutti i segnali. Una volta identificati i segnali presenti nel file, viene eseguita la funzione **calculate_header**(stateful_create.cpp:854) che ha lo scopo di calcolare l'header, i pulse e gli space che identificano l'uno e lo zero e il trailing pulse.

Se fin'ora è andato tutto bene, si passa alla processazione dei file che individuano ciascun pulsante registrato. La funzione che si occupa di questo è **create_lircd_cfg**(stateful_create.cpp:929). Per ogni file, converte i segnali nella rappresentazione binaria, cerca qual'è la posizione che cambia e salva l'informazione che si trova in queste posizioni. La ricerca è abbastanza semplice. Viene eseguito un loop sui bit e controlla se è diverso in almeno un segnale. A questo punto per ogni porzione che cambia, creiamo un oggetto di tipo **struct ir_ncode**(stateful_create.cpp:362) assegnando al campo **code** il valore che si trova in questa posizione (trasformandolo in unsigned long) e specificando nel nome la sua posizione. Questa scelta è stata fatta in quanto più pulsanti potrebbero riferirsi ad una stessa posizione e perciò sarebbe ridondante avere più codici identici per la stessa posizione.

Ad esempio, consideriamo il codice binario "0111" che si trova nel byte numero 8, nei primi 4 bit.

...	0	0	***0***	***1***	***1***	***1***	0	0	0	0	...
...	7	8	***1***	***2***	***3***	***4***	5	6	7	8	...
7			8								...

→

name	$BYTE\_[8\_1\_4]$
code	7
...	...

Tabella 3.2: Esempio contenuto struct ir_ncode

Come avete notato, la temperatura, il timer o qualsiasi altro pulsante

che rappresenta dei valori numerici non viene decodificato, ma viene salvato il valore finale che esso crea. Nel nostro caso, il numero binario "0111" rappresenta 17°C, ma 17 in binario è **0001 0001**, quindi il dispositivo che abbiamo testato prende i 4 bit a destra, scambia 1 con 0, inverte questo valore e poi lo assegna al segnale. Altri dispositivi potrebbero avere meccanismi diversi perciò non esiste un algoritmo unico che funzioni con tutti.

Creati tutti i codici di tutti i file, dobbiamo ordinarli in base alla loro posizione(stateful_create.cpp:520). Questo passaggio è fondamentale e serve a più scopi:

1. Una facile eliminazione dei duplicati
2. Aggiungere eventuali codici non registrati insieme a porzioni del segnale che non cambiano mai.
3. Garantire il corretto funzionamento di *stateful_send* in quanto i codici devono essere ordinati in base alla propria posizione nel segnale.

Siccome i nomi devono essere univoci, e una stessa porzione del segnale può avere più combinazioni possibili, rinominiamo i codici aggiungendo un contatore a rottura della posizione.

```
.............
BYTE_[7_1_4]_1
BYTE_[7_1_4]_2
BYTE_[7_5_8]_1
BYTE_[8_1_4]_1
BYTE_[8_1_4]_2
BYTE_[8_1_4]_3
BYTE_[8_1_4]_4
BYTE_[8_1_4]_5
.............
```

Infine assegnamo i codici ottenuti alla REMOTE creata all'inizio e lanciamo la funzione **fprint_remote** (dump_config.c) di LIRC che scrive tutte le informazioni all'interno del file lircd.conf nel formato da loro definito.

Alla fine dell'elaborazione all'utente vengono forniti i nomi dei "gruppi" creati per ogni file.

```
For file arrow.sa, valid codes are:
BYTE_[8_1_4]
For file mode.sa, valid codes are:
BYTE_[7_1_4]    BYTE_[8_2_4]    BYTE_[9_1_3]
For file power.sa, valid codes are:
BYTE_[6_3_3]
For file sleep.sa, valid codes are:
BYTE_[9_1_1]
For file fan.sa, valid codes are:
BYTE_[9_1_3]
For file swing.sa, valid codes are:
BYTE_[9_4_6]
```

Dall'esempio sopra notiamo che il "gruppo" BYTE_[6_3_3] si riferisce all'accensione del condizionatore. Siccome il pulsante ON/OFF può solo accendere e spegnere, nel file di configurazione ci ritroveremmo 2 codici:

```
BYTE_[6_3_3]_1
BYTE_[6_3_3]_2
```

Per capire quale dei 2 accende e quale spegne, è necessario andare a "tentativi", ovvero usare la funzione stateful_send passando come argomento uno dei 2 codici e verificare cosa succede al proprio condizionatore. Usando la stessa procedura si può capire anche quale codice si riferisce alla temperatura **23°C**, oppure chi è la modalità **COOL**, ecc.

Per vedere tutti i codici creati, o si usa la direttiva **list** di irsend oppure semplicemente si apre il file di configurazione appena creato.

Flowchart

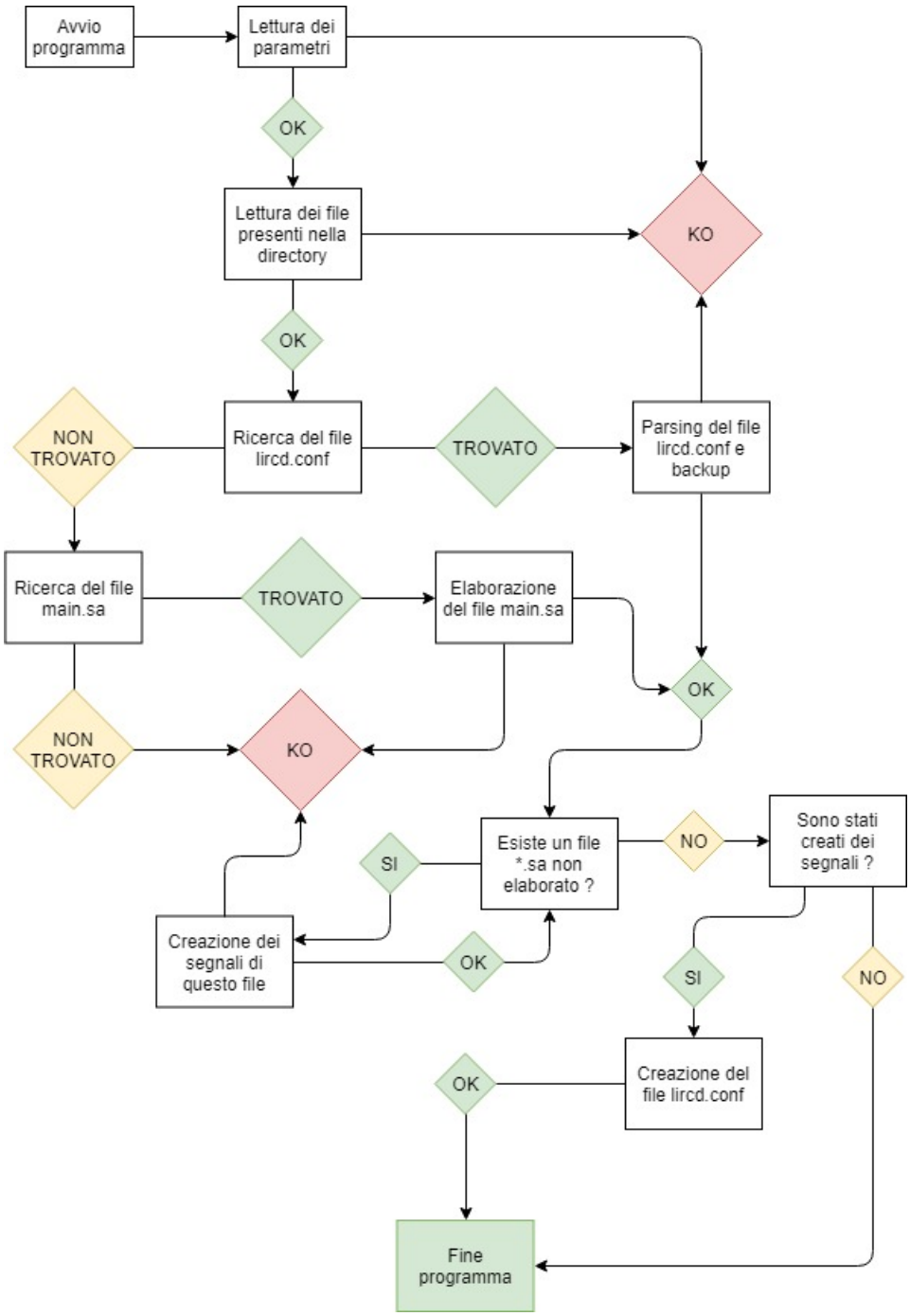

Figura 3.2: Flowchart stateful_create

3.3.2 STATEFUL_SEND

A differenza di stateful_create, questa è una nuova "direttiva" per **irsend**. Il suo scopo è di inviare un segnale unico costruito a partire dai codici presenti all'interno del file di configurazione. In altre parole, questa funzione vede i codici come una parte di un segnale più lungo. Il comando per avviarlo è il seguente:

```
irsend [options] stateful_send <remote> <code> [<code> ... ]
```

Parametri

Le opzioni, come i parametri obbligatori sono esattamente gli stessi che esistevano già. Tuttavia guardiamo quelli obbligatori.

1. `<remote>`

   deve contenere il nome assegnato al momento della creazione del file di configurazione attraverso il parametro **-n**.

2. `<code>`

   deve contenere un nome di un codice presente all'interno del file di configurazione. Possono essere specificati più codici contemporaneamente separati da uno spazio (anche non in ordine). Si può passare anche la stringa vuota per indicare di usare l'ultimo stato inviato o nel caso non esistesse verrà mandato quello predefinito.

Alcuni esempi d'uso:

```
irsend stateful_send TectroAC ""
irsend stateful_send TectroAC BYTE_[8_1_4]_3
irsend stateful_send TectroAC BYTE_[8_1_4]_8 BYTE_[7_1_4]_2
```

Funzionamento

Prima di iniziare, dobbiamo dire cos'è lircd. Come ci indica la 'd' finale del nome, è un daemon, cioè un programma che gira in sottofondo, e che non

prevede iterazioni con l'utente. Affinché irsend funzioni, è necessario avviarlo semplicemente tramite il comando

```
sudo lircd [options]
```

Al momento dell'avvio, va a leggere i file di configurazione all'interno di una specifica directory (come indicato sul sito ufficiale), e gli carica in memoria. Sarà lircd che si occuperà dell'invio del segnale. Il nostro compito è quello di crearlo integro.

Avviato irsend con la nostra direttiva, questo comando non viene eseguito da questa funzione, ma in realtà lo invia al daemon lircd. L'esecuzione di questa direttiva avviene qui dove ad essa è assegnata una funzione (lircd.cpp:239) che per comodità si chiama esattamente come la direttiva. Controlla se esiste un dispositivo assegnato al nome richiesto, dopodiché inizia il parsing dei *"code"* forniti attraverso la funzione **get_stateful_ir_ncode**(lircd.cpp:1585).

Nel caso in cui *"code"* sia "" (carattere blank), allora tutto è facile. Esegue la funzione **ss_get_defaults**(stateful_send.c:488) la quale guarda se esiste uno stato precedente ed in caso affermativo restituisce questo a lircd, altrimenti esegue un loop sui codici presenti internamente (caricati al momento dell'avvio di lircd), creando il segnale usando i primi codici per ogni gruppo.

Nel caso invece in cui, "*code*" sia valorizzato, viene eseguita la funzione **ss_get_dynamics**(stateful_send.c:509). Come sopra inizia un loop sui comandi presenti ed in più, per ogni gruppo controlla se esiste un codice richiesto dall'utente. Se non esiste prende l'ultimo usato (o quello di default la prima volta) altrimenti usa quello richiesto.

Dopo l'invio del segnale, se andato a buon fine viene salvato come ultimo da usare quindi, nell'invio successivo(lircd.cpp:1609).

Vediamo più in dettaglio la creazione del segnale. Come prima cosa conta il numero di comandi richiesti e crea un dizionario (dictionary.c già esiste all'interno di LIRC) ai fini di velocizzare la ricerca del codice richiesto e lo indicheremo con D. Inoltre viene istanziato un array che conterrà l'intera sequenza binaria che chiameremo A. La sua dimensione è indicata nel campo "bits" presente anch'esso all'interno del file di configurazione. A questo punto

viene avviato un loop sui comandi presenti internamente, e per ciascuno di essi fa le seguenti considerazioni:

1. Legge le informazioni presenti tra le parentesi quadre all'interno del campo *name* (il byte di appartenenza, il bit d'inizio e quello di fine).

2. Se la posizione del codice corrente è diversa da quella del codice precedente, verifica se per la posizione precedente abbiamo già trovato un codice, altrimenti inserisce nell'array A il valore binario del campo *code* dell'ultimo codice usato.

3. Se il codice corrente è stato richiesto, ovvero se è presente all'interno del dizionario D, allora prende il valore binario del campo *code* e lo aggiunge all'array A.

4. Va al codice successivo

Finito il loop, viene avviata la funzione, indicata nel campo **checksum_alg** nel file di configurazione, per il calcolo del byte di controllo, dopodiché viene creato un nuovo **struct ir_ncode**(stateful_send.c:179) con le seguenti proprietà:

- *name*: tutti i nomi dei comandi usati separati dalla virgola

- *code*: in questo campo non viene inserito nulla in quanto il tipo long non è abbastanza grande da contenere il valore di questo codice binario.

- *length*:

$$bits \times 2 + 3 \tag{3.1}$$

  Sia "1" che "0" sono rappresentati da un pulse ed uno space, in più va aggiunto il pulse e lo space dell'header assieme al pulse finale.

- *signals*: Questo è un array di interi di dimensione *length*. I primi due valori vengono letti dai campi *phead* e *shead*, dopo di che tramite un loop sull'array A si aggiunge *pone* e *sone* se si incontra un "1" altrimenti *pzero* e *szero*. Infine viene accodato *ptrail*. Tutti questi valori sono scritti all'interno del file di configurazione. Ad esempio:

```
    bits          112
    header       3164  1555
    one           530  1055
    zero          530   290
    ptrail        565
```

Siccome al momento dell'avvio, lircd ha caricato questo file non usando la modalità **RAW**, l'invio non funziona ancora perché tenta di inviare il valore *code* e non *signal*. Bisogna cambiare l'impostazione della struttura ir_remote presente in memoria aggiungendo il flag *RAW_CODES* dicendo così a lircd di usare i valori appena creati.

```
remote->flags |= RAW_CODES;
```

Flowchart

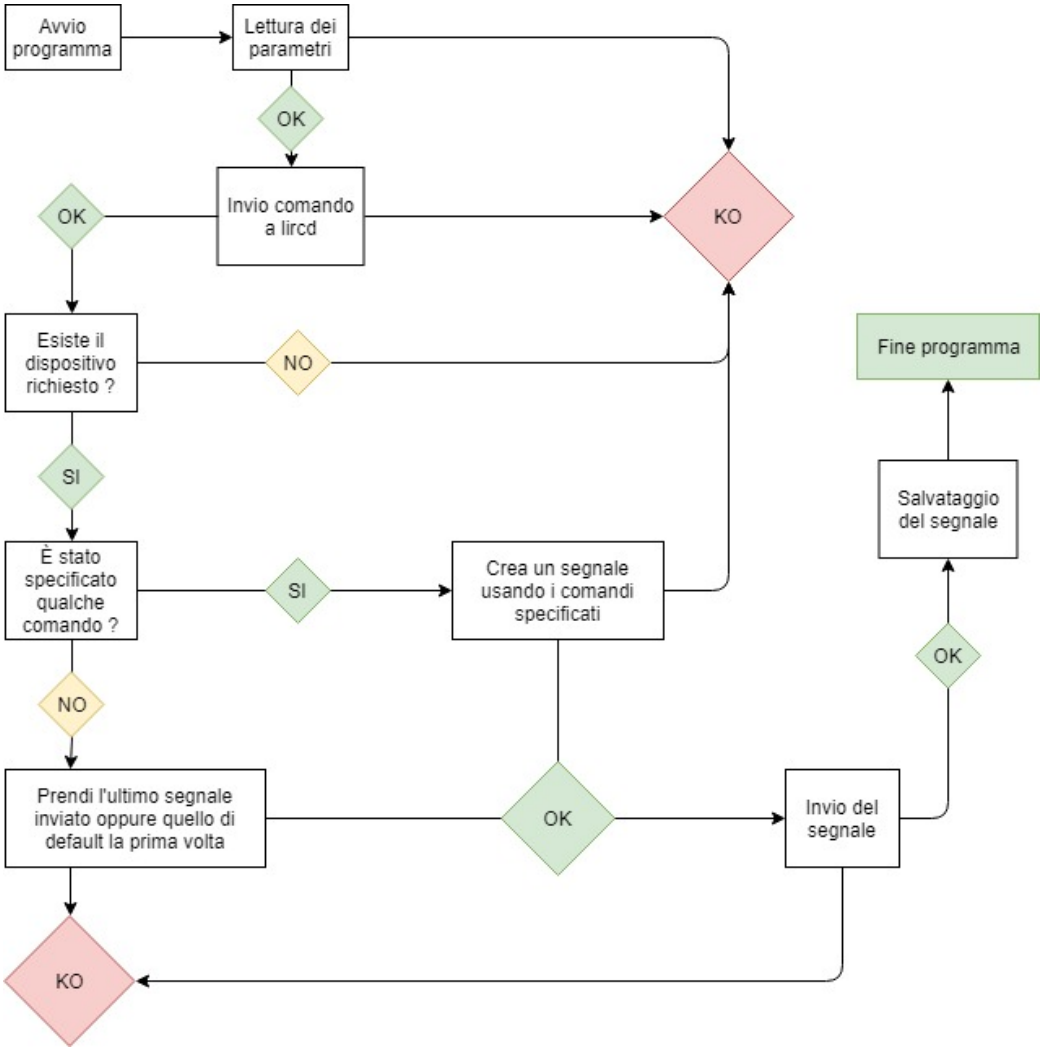

Figura 3.3: Flowchart stateful_send

3.3.3 STATEFUL_GET

Come abbiamo detto in precedenza stateful_send dopo l'invio salva il segnale in un apposito campo che viene mantenuto in memoria. Questo è dovuto al fatto che lircd è un processo sempre attivo. Come stateful_send, anche stateful_get è una direttiva per il tool **irsend** e non fa altro che restituire l'ultimo segnale inviato in output, ovvero lo stato corrente del nostro "telecomando". Il comando per avviarlo è il seguente:

```
irsend [options] stateful_get <remote> ""
```

Parametri

Prende in input solo il nome del dispositivo ed un carattere blank. Quest'ultimo lo abbiamo lasciato obbligatorio solo per non dover modificare ulteriormente i controlli di irsend.

Funzionamento

Analogamente a *stateful_send*, *stateful_get* invia il comando a lircd il quale esegue la funzione assegnata a questa direttiva(lircd.cpp:240). Questa, a sua volta guarda l'ultimo segnale inviato per questo dispositivo e solo se esiste, esegue la funzione **strtok_r**(lircd.cpp:1681) e restituisce in output ogni *token* (ricordiamo il che il nome dell'ultimo segnale inviato contiene tutti i nomi dei codici usati separati dalla virgola). Esempio di output (in realtà non sono 3 colonne, ma una sola):

```
BYTE_[1_1_8]_1      BYTE_[6_4_8]_1      BYTE_[9_1_3]_1
BYTE_[2_1_8]_1      BYTE_[7_1_4]_4      BYTE_[9_4_6]_1
BYTE_[3_1_8]_1      BYTE_[7_5_8]_1      BYTE_[9_7_8]_1
BYTE_[4_1_8]_1      BYTE_[8_1_4]_10     BYTE_[10_1_8]_1
BYTE_[5_1_8]_1      BYTE_[8_2_4]_1      BYTE_[11_1_8]_1
BYTE_[6_1_2]_1      BYTE_[8_5_8]_1      BYTE_[12_1_8]_1
BYTE_[6_3_3]_2      BYTE_[9_1_1]_1      BYTE_[13_1_8]_1
```

Flowchart

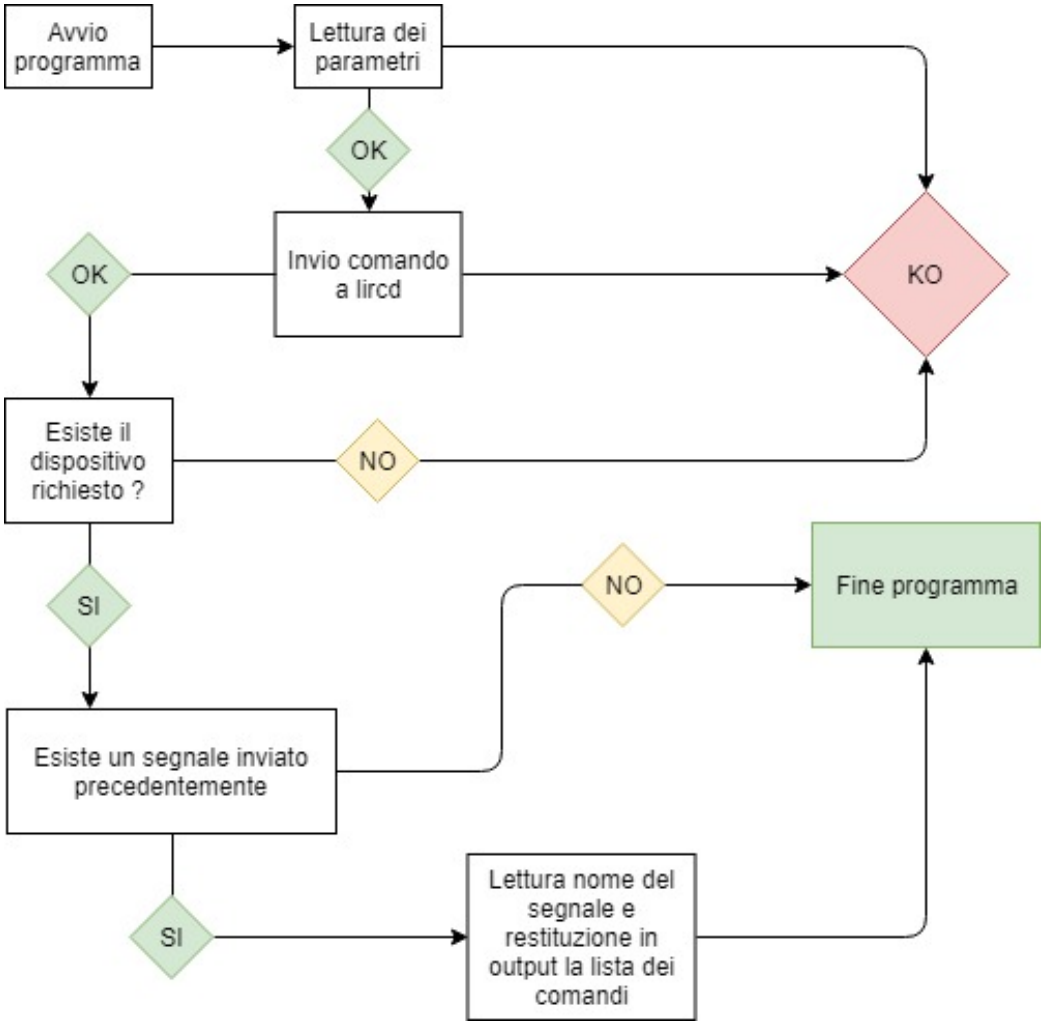

Figura 3.4: Flowchart stateful_get

3.3.4 Aggiungere un nuovo algoritmo per il byte di controllo

Per poter aggiungere un nuovo algoritmo sono necessarie piccole implementazioni in C. Il file dove sono implementati tali algoritmi è **stateful_checksum_alg.c**. Al suo interno ci sono 2 array:

1. inner_algorithms
2. algorithms

Il primo contiene una serie di algoritmi *parziali*, i quali attraverso una combinazione formano uno completo. Essi sono *SUM*, *RIGHT*, *LEFT*, *XOR*, *COUNT1*, *COUNT0*, *FLIP*, *REVERSE*. Il secondo array, che abbiamo descritto nella sezione 3.3.1, contiene semplicemente un elenco di combinazioni del primo. Al momento del calcolo del byte di controllo, l'algoritmo richiesto viene separato dal carattere "_" (underline) e per ogni algoritmo viene eseguita la funzione ad esso assegnata prendendo in input l'output di quello precedente. Solo il primo algoritmo prende in input l'array binario iniziale. Ad esempio, per l'algoritmo REVERSE_SUM_RIGHT_REVERSE, l'esecuzione è la seguente:

```
A = REVERSE(Array Binario)
B = SUM(A)
C = RIGHT(B)
FINAL = REVERSE(C)
```

Infine quest'ultimo risultato viene concatenato all'array binario di partenza e restituito al chiamante.

Per aggiungere un proprio algoritmo perciò, bisogna implementare gli algoritmi parziali mancanti ed aggiungerli rispettivamente agli entrambi gli array.

Conclusioni e Futuri Sviluppi

Dopo aver provato alcuni condizionatori d'aria, siamo riusciti a registrare tutti i pulsanti che volevamo. Sappiamo che nei giorni d'oggi esistono svariati condizionatori d'aria con diversi protocolli e diversi algoritmi di calcolo per il checksum, perciò è impossibile testarli tutti. Tuttavia questo strumento dovrebbe essere in grado di riprodurre un vasta gamma di condizionatori d'aria.

3.4 Automatismo Algoritmo Di Checksum

Al momento le nuove funzionalità richiedono che l'utente conosca il proprio algoritmo per il calcolo del byte di controllo. Sarebbe meglio da implementare invece, un'ulteriore funzionalità che capisca da sola se per il dispositivo che si tenta di registrare esiste o meno un algoritmo per il byte di controllo. Questa potrebbe essere, ad esempio, implementata nel momento in cui si decodificano i vari segnali perché un segnale "intero" contiene anche il byte di controllo.

3.5 Niente File In Input

Un'ulteriore miglioria potrebbe essere di non prendere una directory con n file in input, ma di modificare stateful_create in modo che interagisca con l'utente. Per esempio, potrebbe chiedere all'utente come prima cosa, di premere tutti i pulsanti del telecomando ed eseguire quello che si esegue su

main.sa, dopodiché di premere un pulsante alla volta dandogli un nome ed eseguire quello che si esegue per ogni file *.sa.

3.6 Calcolo Automatico di Stateful_create *-b*

Questo parametro, come abbiamo visto nella sezione 3.3.1, va dato in input. Non è difficile individuarlo nemmeno per un utente inesperto, tuttavia non sarebbe male averlo automaticamente. Esso può essere ottenuto calcolando la differenza tra tutti i pulse e space, dopodiché facendo la media tra quello più grande e quello più piccolo si dovrebbe ottenere un valore valido.

3.7 Posizione Byte di Controllo

Una cosa che abbiamo sottinteso durante questo sviluppo, è che il byte di controllo si trovi in fondo al segnale. Durante lo sviluppo di questa estensione, tutti i condizionatori d'aria valutati avevano sempre il byte di controllo in fondo. Potrebbe comunque non essere sempre vero, perciò un'ulteriore implementazione dovrebbe prevedere una posizione diversa. Questo si può ottenere guardando quale porzione del segnale cambia "sempre", indipendentemente dallo stato del telecomando.

3.8 Supporto altri Protocolli

Questo progetto è valido al momento solo per il protocollo NEC ed altri simili a questo. È necessario invece aggiungere il supporto di molti più protocolli per ampliare la gamma dei dispositivi supportati. Quindi modificare la funzione di decodifica in modo che prima di iniziare a decodificare il segnale, capisca il protocollo da usare. Ad esempio, per il protocollo RC5 il parametro -b della funzione stateful_get non è valido, in quanto il pulse e lo space del "1" sono identici a quello dello 0, solo che sono invertiti (vedi Figura 3.5).

Una soluzione sarebbe quella di definire a priori tutti i protocolli (o almeno quelli più importanti), e poi confrontarli con i segnali ricevuti.

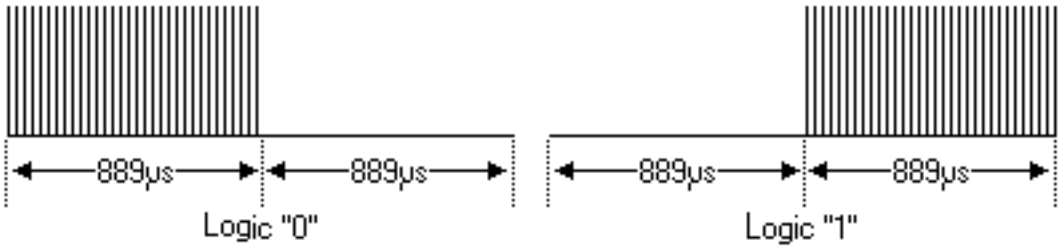

Figura 3.5: Modulazione RC5

Bibliografia

[1] Linux Infrared Remote Control, www.lirc.org

[2] Raspberry Pi, https://www.raspberrypi.org/

[3] Arduino, https://www.arduino.cc/

[4] SB-Projects, https://www.sbprojects.net/knowledge/ir/

[5] Phidgets, https://www.phidgets.com/docs/IR_Remote_Control_Primer

[6] AnalysIR, http://www.analysir.com/blog/

[7] Control your HVAC Infrared Devices from the Internet with IR Remote (Arduino UNO / Raspberry Pi Compatible), https://www.cooking-hacks.com/documentation/tutorials/control-hvac-infrared-devices-from-the-internet-with-ir-remote/

[8] Austin Stanton, Creating A Raspberry Pi Universal Remote With LIRC https://www.hackster.io/austin-stanton/creating-a-raspberry-pi-universal-remote-with-lirc-2fd581

[9] mat_fr, Reverse Engineering Air Conditioner IR Remote Control Protocol, https://www.instructables.com/id/Reverse-engineering-of-an-Air-Conditioning-control/

[10] Bernard Tinkerer, LIRC Node.js Web, https://github.com/bbtinkerer/LircNodeJsWeb

[11] Vishay, Data Formats for IR Remote Control, http://www.vishay.com/docs/80071/dataform.pdf

Ringraziamenti

Dopo lunghi e intensi mesi, finalmente il giorno è arrivato: scrivere queste frasi di ringraziamento è il tocco finale della mia tesi. È stato un periodo di profondo apprendimento sia a livello scientifico nonché personale. Scrivere questa tesi ha avuto un forte impatto sulla mia personalità.

Prima di tutto, vorrei ringraziare la mia famiglia che, con il loro dolce e instancabile sostegno, sia morale che economico, mi hanno permesso di arrivare fin qui oggi, contribuendo alla mia formazione personale.

Ringrazio il professore Renzo Davoli, relatore di questa tesi di laurea, oltre che per l'aiuto fornitomi in tutti questi anni e la grande conoscenza che mi ha donato, per la disponibilità e precisione dimostratemi durante tutto il periodo di stesura. Senza di lui questo lavoro non avrebbe preso vita!

Infine ringrazio gli amici e i compagni di corso che mi hanno generosamente aiutato nelle diverse occasioni.