Corso di programmazione per le scuole con Arduino – PARTE 3

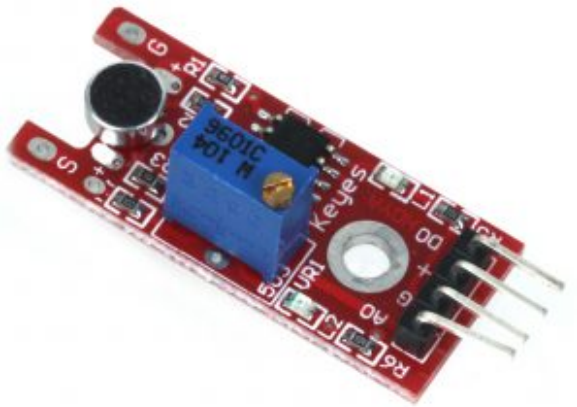
Nella scorsa puntata di questo corso abbiamo presentato una serie di esempi per imparare a programmare con Arduino partendo da zero, rivolti soprattutto a studenti, insegnanti, designer, e altre persone che non sono già abituate a programmare. Vogliamo ora proseguire con degli altri esempi, un po' più avanzati ma comunque abbastanza semplici, spiegandoli nei dettagli. Spiegheremo in particolare come utilizzare un microfono per riconoscere suoni come il battito delle mani o un fischio, e un buzzer (i piccoli altoparlanti a cristallo di quarzo) per suonare delle melodie. Ma spiegheremo anche come controllare un relay, grazie al quale diventa possibile utilizzare Arduino per accendere e spegnere a comando elettrodomestici di vario tipo come lampade o stufe elettriche a 220Volt. Le applicazioni di questi esempi sono quindi valide per insegnare ai bambini delle scuole primarie e secondarie di primo grado la logica di base, ma anche per i designer che vogliono realizzare opere d'arte interattive.

Table of Contents

- 5 Accendere un relay con un microfono
- Distinguere un suono dal rumore di fondo
- 6 Un allarme sonoro collegato ad una porta
- La melodia da suonare
- Solo se il pulsante non è premuto

5 Accendere un relay con un microfono

Il primo esempio che vediamo è molto semplice ma anche molto elegante ed utile. Proveremo, infatti, ad accendere un relay: i relay (o relé) sono dei semplici interruttori che possono accendere (o spegnere) dispositivi alimentati con un alto voltaggio, come la normale 220V delle prese elettriche di casa controllabili con Arduino. Ogni relay ha due pin da collegare ad Arduino (uno al GND e l'altro ad un pin digitale, come un led, ed eventualmente anche un ulteriore pin ai 5V di Arduino), e due pin cui collegare il cavo della corrente (per esempio quello di una lampada, al posto di un normale interruttore). Con Arduino possiamo accendere il relay come se fosse un normale led, semplicemente impostando il valore HIGH sul suo pin digitale. Quindi possiamo decidere di accendere il relay in qualsiasi momento: per esempio, possiamo collegare un microfono ad Arduino (noi ci siamo basati sul semplice ed economico KY-038) ed accendere o spegnere il relay quando viene misurato un valore abbastanza forte (per esempio quando qualcuno batte le mani vicino al microfono).



Il microfono KY-038 per Arduino

Il codice del programma da scrivere nell'Arduino IDE e

caricare sulla scheda comincia con la classica dichiarazione delle variabili:

Prima di tutto indichiamo i pin che stiamo utilizzando: la variabile **relay** conterrà il numero del pin digitale cui abbiamo collegato il relay, mentre **sensorPin** contiene il numero del pin analogico cui abbiamo collegato il segnale del microfono. I microfoni, infatti, sono sensori analogici.

Nella variabile **sensorValue** inseriremo il valore letto dal sensore, che quindi sarà rappresentato da un numero compreso tra 0 e 1023 perché questo è l'intervallo dei sensori analogici.

Definiamo poi una variabile di tipo **bool**, ovvero booleano. Una variabile booleana può avere due soli valori: vero o falso, **true** o **false** in inglese. La utilizzeremo per memorizzare l'attuale stato del relay, e infatti la chiamiamo **on**. Se la variabile **on** è **true** vuol dire che il relay è acceso, altrimenti è spento.

La funzione **setup**, eseguita una sola volta all'avvio di Arduino, predispone il pin digitale cui è collegato il relay in modalità di **OUPUT**.
Sfruttando la funzione **digitalWrite** si può quindi scrivere il valore iniziale del relay, che sarà **LOW**, ovvero spento.

Dove trovare i componenti?

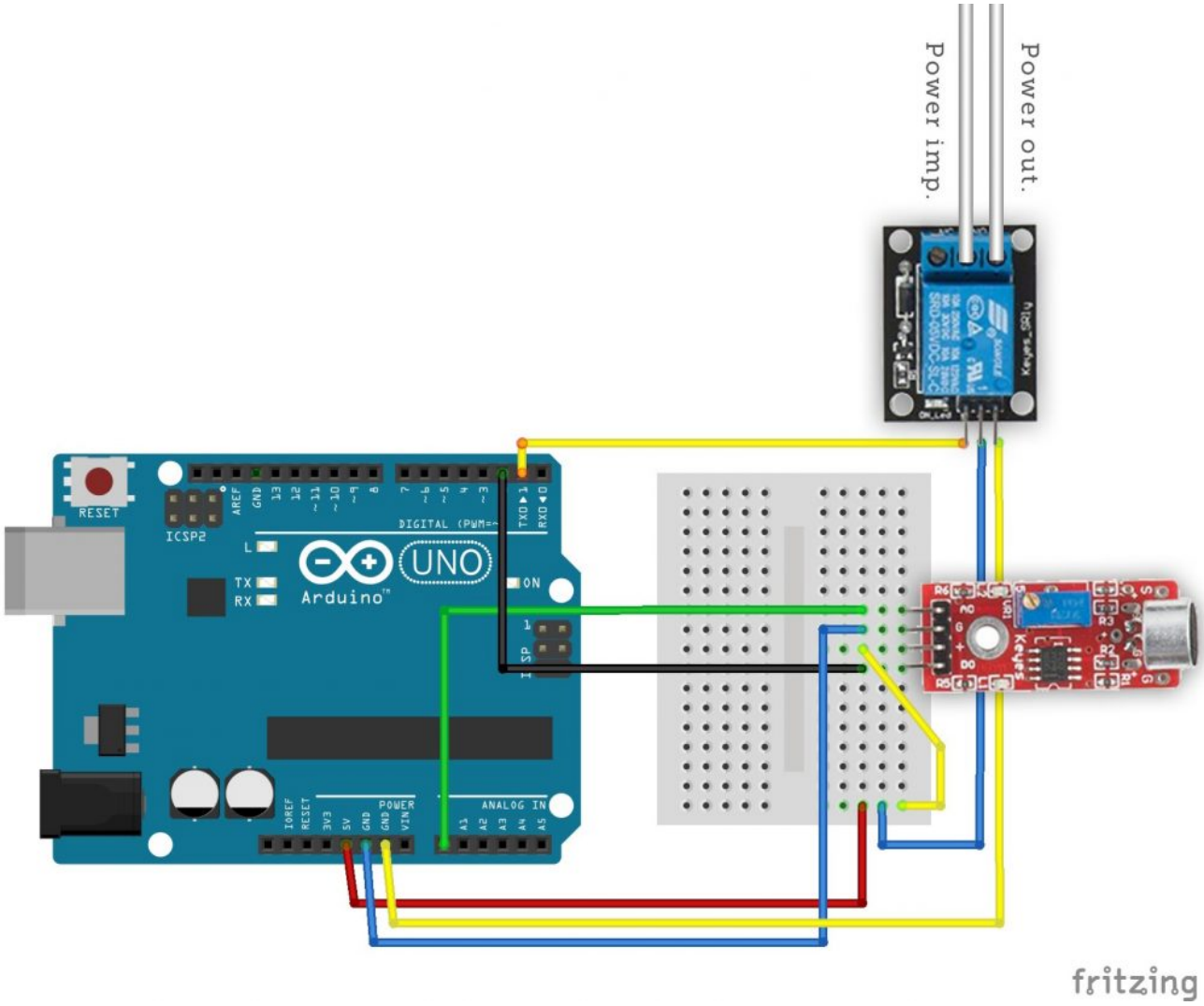
Negli esempi proponiamo dei sensori e degli altri componenti specifici. Mentre un relay si può acquistare in qualsiasi negozio di elettronica, ed anche un pulsante od un buzzer, il microfono e la scheda con i vari sensori infrarossi sono più

rari. Ma possiamo trovare tutti questi componenti, eventualmente indicando le sigle che abbiamo suggerito (KY-038), su ebay e su Aliexpress a prezzi molto bassi. Bisogna solo prestare attenzione alle diverse versioni disponibili: spesso, le schede che costano meno richiedono ancora qualche saldatura, mentre ne esistono altre, che costano pochi euro in più, già dotate di connettori di semplice utilizzo.

Poi abilitiamo anche la porta seriale, così sarà possibile scrivere un messaggio al computer eventualmente collegato ad Arduino per fargli sapere cosa stiamo misurando con il microfono.

La funzione **loop** viene ripetuta all'infinito finché Arduino è acceso, quindi è quella che utilizziamo per realizzare le attività del nostro progetto. Per cominciare, a ogni ciclo provvediamo a leggere l'attuale valore del microfono, che ovviamente è un numero compreso tra 0 e 1023 a seconda del volume percepito dal microfono, memorizzandolo nella variabile **sensorValue**.

Ora possiamo scrivere il numero ottenuto sulla porta seriale, così possiamo controllarlo con un computer. Leggere il valore può essere utile per capire se il microfono debba essere regolato (di solito c'è una apposita rotella) in modo da non ottenere numeri troppi alti o troppo bassi.



Tipico collegamento di microfono e relay ad Arduino

Distinguere un suono dal rumore di fondo

Ora dobbiamo decidere la soglia oltre la quale consideriamo il suono registrato dal microfono adeguato a causare l'accensione o lo spegnimento del relay.

Un semplice **if** ci permette di risolvere il problema, e possiamo scegliere qualsiasi valore come soglia: noi abbiamo scelto 500, ma potete alzarlo o abbassarlo per vedere cosa funziona meglio con il vostro microfono. Se la soglia è stata superata, quindi è stato percepito abbastanza rumore, dobbiamo

agire sul relay. Ma come? Semplice: se il relay è acceso lo vogliamo spegnere, se invece è spento lo vogliamo accendere. In poche parole, dobbiamo invertire il valore della variabile **on**, che indica l'attuale stato di accensione del relay, e che dovrà passare da **false** a **true** e viceversa. Possiamo farlo con l'operatore logico **NOT**, ovvero il punto esclamativo. In poche parole, se **on** è **false**, **!on** sarà **true** e viceversa. Quindi scrivendo **on = !on** abbiamo semplicemente detto ad Arduino di invertire il valore della attuale variabile **on**, scegliendo il suo contrario.

Ora possiamo scrivere l'attuale valore della variabile **on**, sia esso true o false, sul pin digitale del relay. Infatti, in Arduino il valore **true** corrisponde al valore **HIGH**, mentre il valore **false** corrisponde al valore **LOW**. Quindi, avendo una variabile di tipo **bool**, possiamo assegnare direttamente il suo valore ad un pin digitale.

Prima di concludere la funzione **loop**, e il programma, chiamiamo la funzione **delay**, che si occupa solo di attendere un certo numero di millisecondi prima che la funzione **loop** possa essere ripetuta. Abbiamo scelto di attendere 100 millisecondi, vale a dire 0,1 secondi, perché è il tempo minimo per assicurarsi che un rumore rapido come il battito di due mani sia effettivamente terminato, e non venga contato erroneamente due volte. Per essere più sicuri di non commettere errori, possiamo aumentare questo tempo a 1000 millisecondi o anche di più. Il programma è ora completo.
Per spiegare in modo più preciso il funzionamento dell'operatore logico **!**, chiariamo che la riga di codice **on = !on** equivale al seguente **if-else**:

La singola riga di codice che abbiamo utilizzato rende il programma più semplice e più elegante, ma ha esattamente lo stesso significato e lo stesso risultato.

Attenzione alla corrente

Quando lavoriamo con Arduino, stiamo lavorando con l'elettronica, e dunque con della corrente. Ma si tratta di corrente continua a basso voltaggio. Quando aggiungiamo un relay le cose cambiano, perché stiamo andando ad utilizzare anche la normale corrente alternata a 220V delle prese di casa. Ed è molto pericoloso. Le saldature devono essere fatte bene, per evitare possibili cortocircuiti, e non si devono mai toccare contatti scoperti finché la corrente è in circolo. Non dovrebbe mai essere permesso a minorenni di toccare cavi preposti alla conduzione della corrente ad alto voltaggio, anche quando tali cavi siano scollegati dalla presa a muro. Comunque, è bene assicurarsi di lavorare dietro un salvavita, così un eventuale scarica di corrente verrebbe interrotta prima di fulminare un malcapitato. Realizzando questi esempi a scuola conviene sostituire la corrente 220V con un semplice alimentatore da 12V: oggi si trovano molti led che possono essere alimentati direttamente a 12 Volt, riducendo il rischio di farsi male. Il concetto rimane comunque lo stesso, visto che un relay da 220V può tranquillamente essere usato per la corrente 12V continua o alternata.

6 Un allarme sonoro collegato ad una porta

Come si costruisce un sistema di allarme? L'idea è di base è molto semplice, tutto quello che serve è un sensore che rilevi una intromissione, ed un dispositivo sonoro come un altoparlante od un buzzer. Per il nostro esempio sceglieremo un buzzer, anche noto come cicalino o altoparlante piezoelettrico. Ha infatti alcuni vantaggi: è molto economico,

è molto piccolo, e funziona con pochissima corrente quindi non richiede alcuna amplificazione. Per quanto riguarda il sensore, dipende molto da come vogliamo progettare il sistema anti intrusione. L'idea è di controllare una porta: vogliamo un meccanismo che suoni quando una porta viene aperta, e che invece rimanga in silenzio se la porta è chiusa (concettualmente, un po' come la luce del frigorifero, oppure i bigliettini di auguri con la canzoncina).

Un sensore reed può essere recuperato dai contachilometri per biciclette

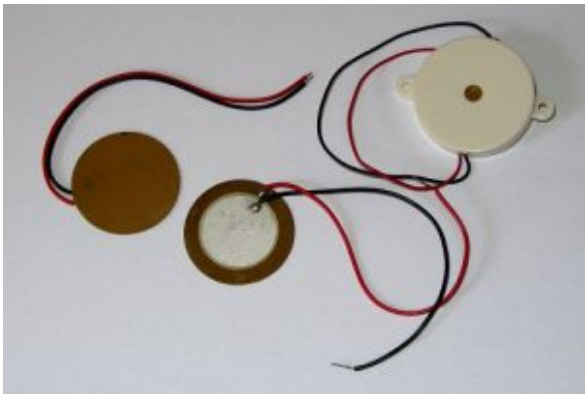
Per questo scopo, in realtà andrebbe bene anche un pulsante: lo si posiziona tra la porta e il muro, così finché la porta è chiusa il pulsante è premuto, mentre appena si apre il pulsante non sarà più premuto. Naturalmente, un pulsante può essere costituito con due pezzi di alluminio (anche quello da cucina in fogli), uno posizionato sulla porta e l'altro sul muro in modo da farli contattare quando la porta è chiusa. In alternativa, si può fissare al muro un sensore reed (anche chiamato reed switch), ed alla porta una calamita: si tratta dello stesso tipo di sensore con cui funzionano i contachilometri per biciclette.
Il codice, che possiamo scrivere direttamente nell'Arduino IDE, è il seguente:

Non servono librerie particolari: tutto ciò che offre il

normale Arduino è più che sufficiente. Però dobbiamo prima di tutto definire le frequenze delle note musicali che ci servono (i valori di riferimento si trovano su https://it.wikipedia.org/wiki/Notazione_dell%27altezza). Per farlo, potremmo dedicare ad ogni nota una variabile, con il valore numerico della frequenza della nota in questione. Ma questo riempirebbe la memoria RAM di Arduino, che è poca. Sarebbe meglio scrivere questi numeri nella memoria flash di Arduino, quella che ospita il codice del programma che carichiamo, perché è molto più grande. Possiamo farlo definendo non una variabile ma una costante, una **costante globale** a essere precisi, con il comando **#define**. In questo modo, per esempio, stabiliamo che la costante **Do4** ha il valore 261, perché la nota do della quarta ottava ha una frequenza di **261 Hertz**.

Definiamo anche una particolare nota con frequenza pari a zero, che utilizzeremo per costituire le pause della musica.

Ora il programma può procedere come al solito, dichiarando due variabili che rappresentino i pin cui sono collegati i componenti elettronici. In particolare, abbiamo deciso di collegare il pulsante (o sensore **reed**) al **pin digitale 2**, ed il **buzzer** al **pin digitale 9**. Il buzzer deve essere collegato ad un pin digitale PWM, indicato dal simbolo ~ sulla scheda Arduino.



Tre buzzer (o cicalini): si

può notare quanto piccoli siano

La melodia da suonare

Ora dobbiamo scrivere le note della musica che vogliamo suonare: le note sono rappresentata da due valori, altezza e durata.

Quindi realizzeremo due diversi **array** (o vettori): un array è, semplicemente una variabile speciale che può contenere molti valori. Praticamente, una lista di valori, ciascuno dei quali potrà poi essere identificato con un numero progressivo tra parentesi quadre, partendo dallo zero. Per esempio, l'elemento **melody[0]** è **La4**, e anche l'elemento **melody[1]** è **La4**, ma l'elemento **melody[2]** è **Si4**.
È possibile costruire array a due dimensioni, praticamente delle tabelle con molte colonne, ma occupano molta memoria e sono scomodi. Meglio dedicare due array diversi alle due diverse informazioni: semplicemente creiamo una lista per l'**altezza** delle note, chiamata **melody**, ed una lista della **durata** delle note chiamata **beats**. Per semplicità, decidiamo che la durata delle note viene indicata come la frazione della nota semibreve. Quindi, se scriviamo 4 intendiamo dire un quarto (ovvero una semiminima), se scriviamo 8 intendiamo un ottavo (una croma), e così via. Naturalmente, si possono anche indicare valori come un terzo o un quinto: non è certo obbligatorio utilizzare numeri pari.

Le note musicali

Arduino è in grado di produrre, con un buzzer tutte le

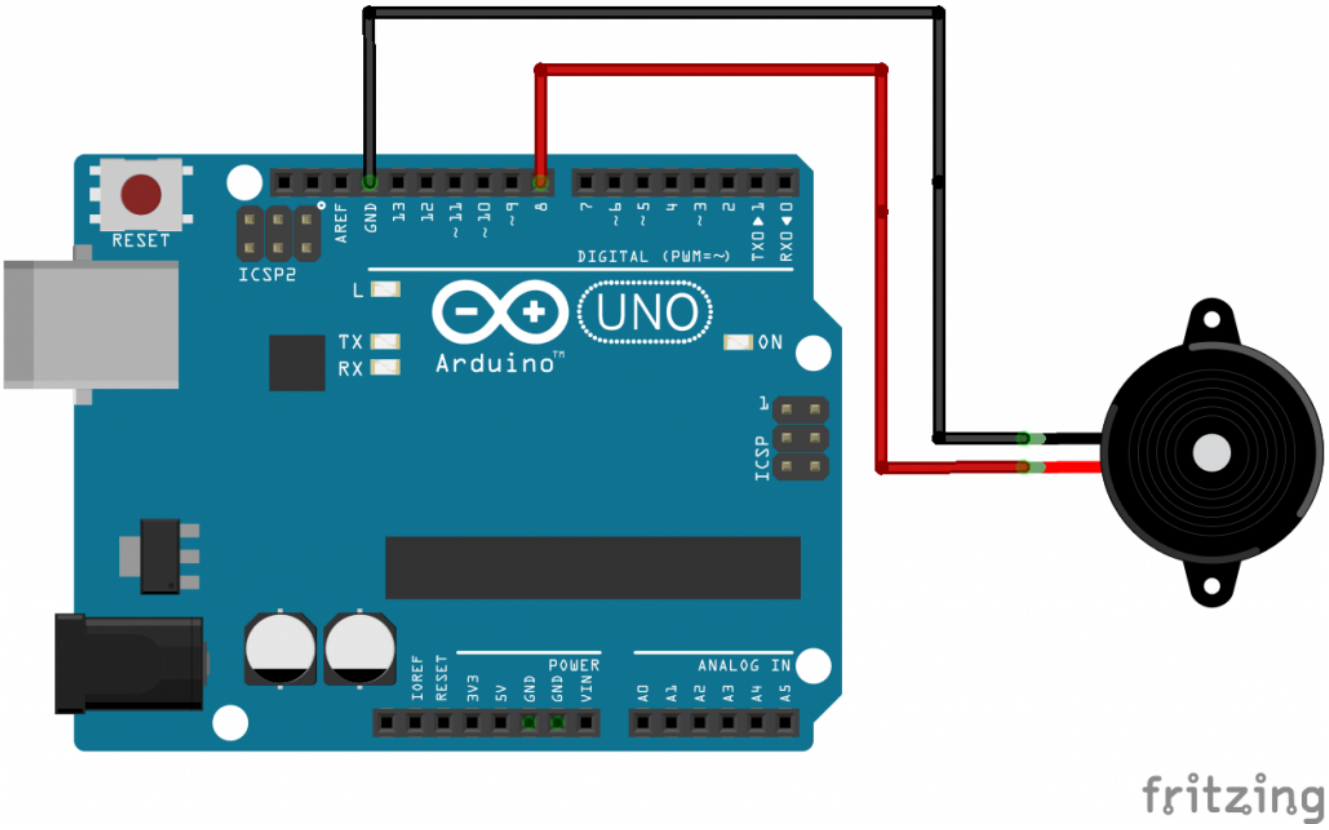
frequenze audio udibili da un essere umano, ed anche alcuni ultrasuoni udibili soltanto da altri animali (come i cani). Le note musicali non sono altro che specifiche frequenze. Le note sono in totale 12, considerando anche i bemolle ed i diesis, e vengono divise in 8 o 9 ottave. Un pianoforte, comunque, non supera l'ottava numero 8. La nota "standard" è il La della quarta ottava, chiamato anche La4, fissato a 440Hz. Il La3 ha una frequenza di 220Hz, mentre il La5 ha una frequenza di 880Hz, e così via.
https://it.wikipedia.org/wiki/Notazione_dell%27altezza

Certo, l'inno alla gioia non è proprio la musica migliore per un allarme. Ma almeno è facilmente riconoscibile, e del resto stiamo realizzando questo esempio per i bambini. Naturalmente, si può scrivere qualsiasi spartito musicale, basta conoscere le note e la loro durata.

Per leggere tutta la melodia dovremo scorrere i due array. E per farlo abbiamo bisogno di sapere quanti elementi sono contenuti nell'array **melody**. Non esiste un metodo diretto per sapere quanti elementi sono presenti dentro un array, però si può utilizzare un trucco: la funzione **sizeof** ci dice la dimensione in byte dell'array. Siccome il nostro array è di tipo **int**, ovvero ogni suo elemento è un numero intero, e in Arduino Uno (con processore a 16 bit) i numeri interi vengono memorizzati in **2 byte**, è ovvio che dividendo la dimensione dell'array per 2 otteniamo il numero di elementi dell'array. Arduino Due ha un processore a 32 bit, quindi per memorizzare i numeri interi utilizza 4 byte invece di due (ogni byte è composto da 8 bit). In quel caso basta dividere per 4 invece che per 2.

Dichiariamo ora due variabili importanti per stabilire la durata generale delle note. La variabile **tempo** contiene, in millisecondi, la durata di una nota semibreve: l'abbiamo impostata a 4000 perché di solito la semibreve viene suonata

in 4 secondi, ma se volete potete modificare l'impostazione per rendere il tutto più veloce o più lento. La variabile **pause**, invece, contiene il tempo che intercorre tra una nota e l'altra: i computer come Arduino sono capaci di suonare le note una dopo l'altra senza alcuna pausa, ma in questo modo si ottiene un suono poco naturale.



Un buzzer può essere collegato ad Arduino connettendo uno dei pin al GND e l'altro ad un pin digitale

Quando a suonare è un essere umano, infatti, c'è sempre una piccolissima pausa tra una nota e l'altra (per esempio il tempo necessario a spostare le dita). Però si tratta di un tempo molto piccolo, quindi lo esprimiamo non in millisecondi, ma in **microsecondi**: 1000 microsecondi sono un millesimo di secondo. È un tempo apparentemente insignificante, ma noterete che senza di esso diventa difficile distinguere il suono delle varie note della melodia.

Ci servono poi tre variabili, che utilizzeremo per capire

quale sia la nota da suonare volta per volta: potremmo dichiararle già nella funzione **loop**, ma lo facciamo nella parte principale del programma così potranno eventualmente essere utilizzate anche in altre funzioni, se qualcuno vorrà migliorare il programma. La variabile **tone_** conterrà la frequenza da suonare, mentre **beat** conterrà la durata della nota espressa come frazione della semibreve. Però Arduino non sa cosa sia la durata di una nota, quindi dobbiamo tradurre la durata delle varie note in millisecondi, ed è quello che faremo con la variabile **duration**. Una nota: la variabile **tone_** non è stata chiamata soltanto tone perché "tone" è anche il nome di una funzione, ed i nomi delle variabili che creiamo devono essere diversi da quelli delle funzioni già fornite da Arduino.

La funzione **setup**, eseguita una sola volta all'accensione di Arduino, non fa altro che attivare i due pin digitali: quello del pulsante sarà un pin di tipo **INPUT**, mentre quello del buzzer sarà ovviamente di tipo **OUTPUT**.

Come funziona un pulsante

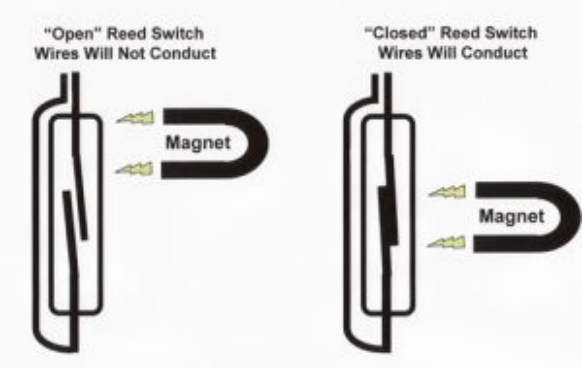
Un pulsante non è altro che un contatto di qualche tipo che ha due posizioni: aperto o chiuso. Quando il contatto è aperto non c'è passaggio di corrente, quando il contatto è chiuso la corrente passa. Un pulsante si costruisce facilmente con Arduino: basta inserire una resistenza da 10KOhm nel pin 5V, ed un cavetto nel pin GND. Il capo libero del cavo va poi collegato al capo libero della resistenza, ed a questa unione va aggiunto un ulteriore cavetto, al quale rimane un capo libero. Quest'ultimo capo libero è uno dei due contatti del pulsante. L'altro contatto si ottiene semplicemente inserendo un cavo in uno dei pin digitali (o analogici, se si vuole) di Arduino, mantenendo libero l'altro capo di questo cavo.

Abbiamo quindi due cavi con un capo libero ciascuno: quando questi si toccano, il pulsante è chiuso, quando non si toccano è aperto. Poi possiamo collegare ai due cavi qualsiasi cosa: un pulsante, un interruttore, un reed, o semplicemente due pezzi di alluminio facendo in modo che volte si tocchino ed a volte no (per esempio fissandoli sul lato di una porta e sul muro).
https://www.arduino.cc/en/Tutorial/Button

Solo se il pulsante non è premuto

La funzione loop è quella che viene eseguita di continuo, quindi è qui che scriveremo il vero e proprio codice "operativo" del programma.

Prima di tutto, dobbiamo occuparci del pulsante: non dimentichiamo che l'idea è di far suonare la melodia solo se il pulsante non è premuto, perché finché è premuto significa che la porta è chiusa. Praticamente, il contrario di un normale pulsante (è come un citofono che suona quando non è premuto invece di suona alla pressione del pulsante).



Un sensore reed è di fatto un pulsante attivato da una calamita

I pulsanti sono fondamentalmente dei sensori digitali, ed offrono ad Arduino due possibili valori: **LOW** (cioè 0 Volt), se il pulsante non è premuto, e **HIGH** (cioè 5 Volt) se il pulsante è premuto. Siccome vogliamo che tutto ciò che segue d'ora in poi avvenga solo se il pulsante non è premuto, controlliamo lo stato del pulsante con la funzione **digitalRead** e verifichiamo grazie a un **if** che tale stato sia uguale a **LOW**.

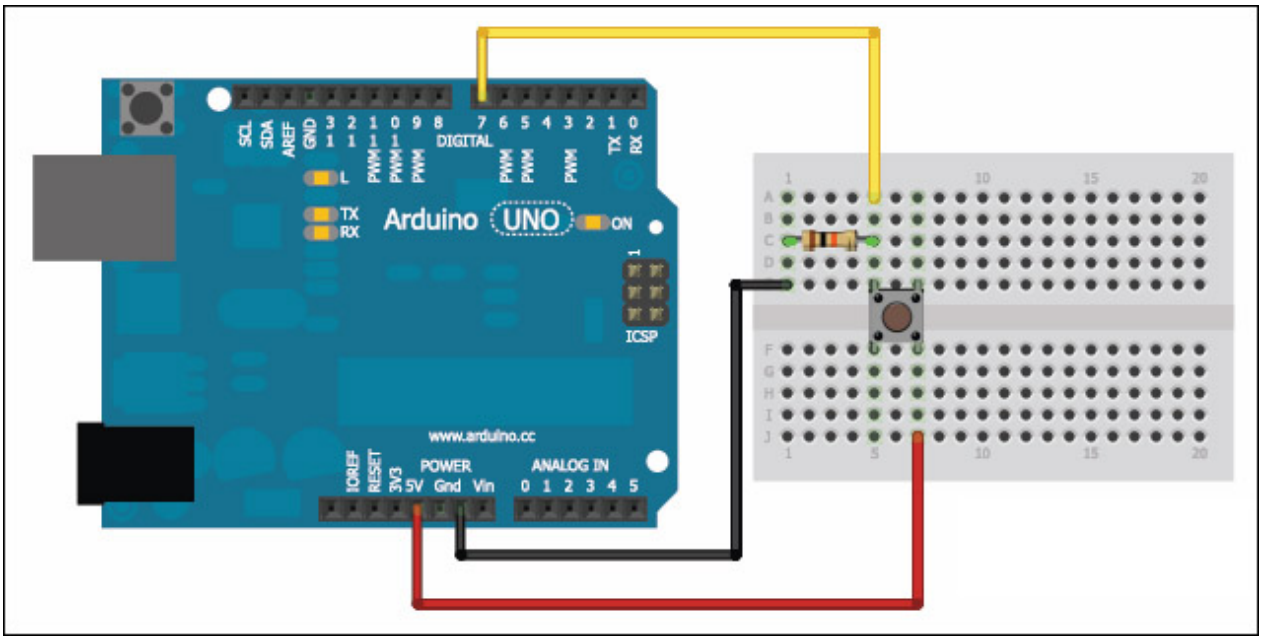
Se il pulsante non è premuto, dobbiamo cominciare a scorrere i due array per leggere le varie note e suonarle. Possiamo farlo con un ciclo **for**: i cicli for hanno una variabile contatore: nel nostro caso la variabile **i**. Il valore iniziale della variabile **i** è **0**, e ad ogni ciclo il suo valore verrà incrementato di **1**, perché questo è previsto dall'ultimo argomento del ciclo for (**i++** infatti significa che ad i va sommato il valore 1). Il ciclo andrà avanti finché la variabile **i** avrà un valore inferiore alla variabile **MAX_COUNT**, che avevamo utilizzato per calcolare il totale degli elementi dell'array. In altre parole, il ciclo comincia con **i** uguale a **0** e termina quando sono stati letti tutti gli elementi dell'array.

Visto che la variabile **i** scorre tutti i numeri dallo **0** al totale degli elementi degli array, possiamo proprio utilizzarla per leggere i vari elementi uno dopo l'altro. Infatti, **melody[i]** è l'i-esimo elemento dell'array che contiene le frequenze delle note, mentre **beats[i]** è l'i-esimo elemento dell'array che contiene le durate delle note. Inseriamo questi valori nelle due variabili che avevamo dichiarato poco fa appositamente. Calcoliamo anche la durata in secondi della nota attuale, inserendo il valore calcolato nella variabile **duration**.

Ora ci sono due opzioni: la frequenza della nota può essere zero o maggiore di zero. Una frequenza pari a zero (o comunque non maggiore di zero) indica una pausa, quindi non si suona

niente, basta aspettare. Un semplice **if** ci permette di capire se la frequenza della nota attuale sia maggiore di zero e quindi si possa suonare.

In caso positivo eseguiamo comunque un controllo: se, infatti, nel bel mezzo della riproduzione il pulsante viene premuto di nuovo (quindi il suo valore diventa HIGH), dobbiamo interrompere la riproduzione della melodia. E per interromperla basta terminare improvvisamente il ciclo for che sta scorrendo tutte le note della melodia.



Il tipico collegamento di un pulsante ad Arduino prevede l'uso di una resistenza da 10KOhm

Questa interruzione può essere fatta con il comando break, che blocca il ciclo for ed esce da esso, facendo sì che il programma termini anche la funzione loop e provveda a ripeterla da capo. Se non si vuole interrompere la riproduzione del suono dopo che essa è già iniziata una volta, basta rimuovere questa riga di codice.

È arrivato, finalmente, il momento di suonare la nota attuale: prima di tutto ci si assicura che il buzzer non stia suonando altre note, per evitare sovrapposizioni. E lo si può fare

chiamando la funzione noTone, indicando il pin digitale cui è collegato il buzzer (nel nostro caso speakerOut). Poi possiamo suonare la nota chiamando la funzione tone: questa richiede tra gli argomenti il pin digitale del buzzer, la frequenza (che è contenuta nella variabile tone_), e la durata della nota. Una cosa interessante della funzione tone è che non blocca il programma fino al termine: vale a dire che anche se noi chiediamo di eseguire una nota di 2 secondi, Arduino non aspetta che il suono della nota sia terminato per procedere con la riga di codice successiva. Questo è molto utile nel caso si abbiano più buzzer collegati ad Arduino e si vogliano suonare contemporaneamente. Però, nel nostro caso, dobbiamo dire ad Arduino di aspettare che la nota sia terminata prima di procedere. Quindi utilizziamo la funzione delay per chiedere ad Arduino di attendere un numero di millisecondi pari proprio alla durata prevista della nota musicale. Inoltre, attendiamo anche un numero di microsecondi pari alla pausa tra una nota e l'altra che avevamo deciso, con la funzione delayMicroseconds.

Come avevamo detto, è possibile che la nota da "suonare" abbia frequenza pari a zero: ed in questo caso è una pausa, quindi non si suona nulla, basta attendere il tempo necessario con la funzione delay. Siccome prima avevamo cercato di distinguere le note vere dalle pause utilizzando un ciclo if, ora basta aggiungere un else per dire ad Arduino cosa fare quando trova una pausa. Terminato anche l'else, possiamo chiudere anche il ciclo for, il ciclo if iniziale (quello che verificava che il pulsante non fosse premuto), e la funzione loop. Il programma è terminato.

Il codice completo

Potete trovare il codice sorgente dei vari programmi

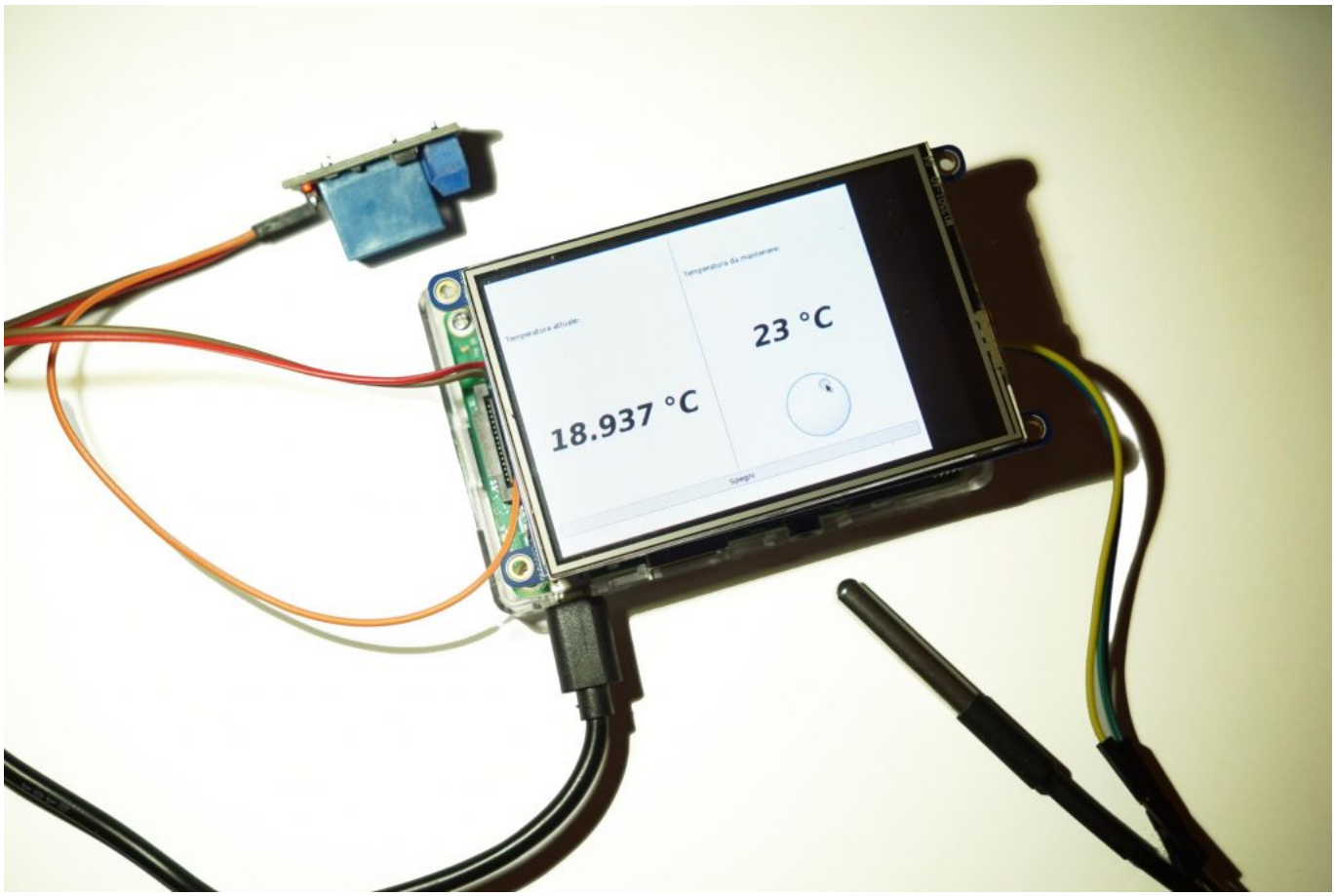
presentati in questo corso di introduzione alla programmazione con Arduino nel repository: https://www.codice-sorgente.it/cgit/arduino-a-scuola.git/tree/ I file relativi a questa lezione sono, ovviamente, **microfono-relay.ino** e **allarme.ino**.

Un termostato touchscreen con RaspberryPi

I RaspberryPi sono una ottima piattaforma su cui costruire oggetti basati sull'idea dell'Internet of Things: dispositivi per uso più o meno domestico che siano connessi a reti locali o ad internet e possano essere facilmente controllati da altri dispositivi. Il vantaggio di un RaspberryPi rispetto a un Arduino è che è più potente, e permette quindi una maggiore libertà. E non solo in termini di collegamento al web, ma anche per quanto riguarda le interfacce grafiche. È un dettaglio del quale non si parla molto, perché tutti danno per scontato che un Raspberry venga usato solo come server, controllato da altri dispositivi con una interfaccia web, e non collegato a uno schermo. Ma non è sempre così. Per esempio, possiamo utilizzare un RaspberryPi per realizzare un termostato moderno. In questo caso non abbiamo più di tanto bisogno di accedervi dallo smartphone: sarebbe utile, ma di sicuro non è l'utilizzo principale che si fa di un termostato. Basterebbe avere uno schermo touchscreen, con una bella grafica, che si possa fissare al muro per regolare la temperatura. Il punto su cui molti ideatori dell'IoT perdono parte del proprio pubblico è il mantenere le cose "con i piedi per terra". La gente, infatti, è abituata a regolare la

temperatura della propria casa da un semplice pannello attaccato al muro, ed è questo che vuole. Magari un po' più futuristico e gradevole alla vista, ma comunque non troppo diverso da quello a cui si è già abituati. Non tutto ha bisogno di essere connesso al web, soprattutto considerando che è un pericolo: se il nostro termostato è raggiungibile da internet, prima o poi un pirata russo si divertirà ad abbassare la temperatura di casa nostra fino a farci raggiungere un congelamento siberiano. La strada più semplice e immediata, a volte, è la migliore. La domanda a questo punto è: come si realizza una interfaccia grafica per Raspberry? Esistono diverse opzioni, ovviamente, ma quella che abbiamo scelto è PySide2. Si tratta della versione Python delle librerie grafiche Qt5, le più comuni librerie grafiche cross platform. È l'opzione migliore semplicemente perché sono disponibili per la maggioranza delle piattaforme e dei sistemi operativi esistenti, quindi i progetti che realizziamo con queste librerie possono funzionare anche su dispositivi differenti dal Raspberry, e ciascuno può adattare il codice alle propri esigenze con poche modifiche.

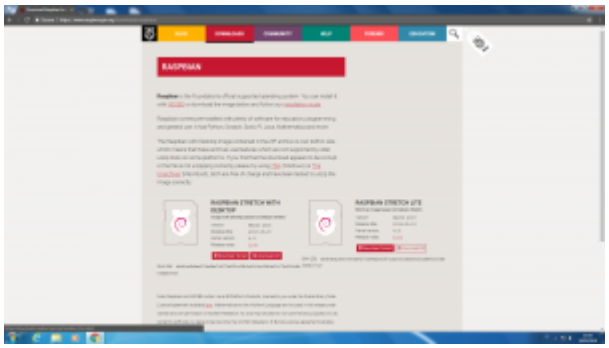
Il nostro dispositivo di riferimento sarà un RaspberryPi 3 B+, dal costo di 50 euro, con uno schermo TFT di Adafruit da 3.5 pollici. Come sistema operativo, si può utilizzare la versione di **Raspbian Buster con PySide2** preinstallato realizzata per Codice Sorgente (https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/).



Ma le procedure presentate possono funzionare anche per il Raspberry Pi Zero W, con lo stesso schermo, bisogna solo aspettare che venga pubblicata la versione Buster di Raspbian per questo dispositivo (prevista tra qualche mese).

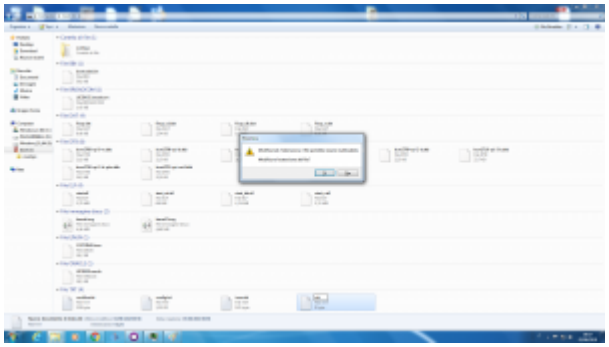
Preparare la scheda sd

Per prima cosa si scarica l'immagine di Raspbian: al momento si può recuperare dal sito ufficiale la versione Raspbian Stretch per qualsiasi tipo di Raspberry. Se invece avete un Raspberry Pi 3 o 3B+ (o anche un Raspberry Pi 2) potete utilizzare **la versione che ho realizzato personalmente di Raspbian Buster, con le librerie Qt già installate: https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/**. Questa è la strada consigliabile, visto che Raspbian Stretch è molto datato e non è possibile installare le librerie Qt più recenti, su cui si basa il progetto di questo articolo.



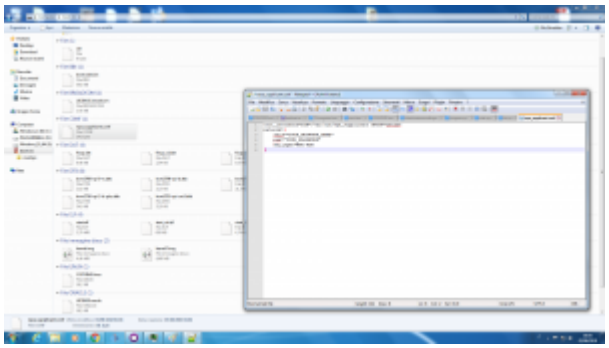
Ottenuta l'immagine del sistema operativo, la si scrive su una scheda microSD con il programma Win32Disk Imager:

Dopo il termine della scrittura, si può inserire nella scheda SD il file **ssh**, un semplice file vuoto senza estensione (quindi ssh.txt non funzionerebbe):

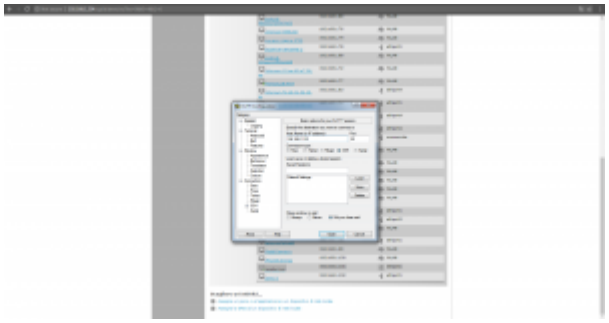


Sempre nella scheda SD si può creare il file di testo **wpa_supplicant.conf**, inserendo un testo del tipo

Bisogna però assicurarsi che il file abbia il formato Unix per il fine riga (LF, invece del CRLF di Windows). Lo si può stabilire con Notepad++ o Kate, degli editor di testo decisamente più avanzati di Blocco Note:



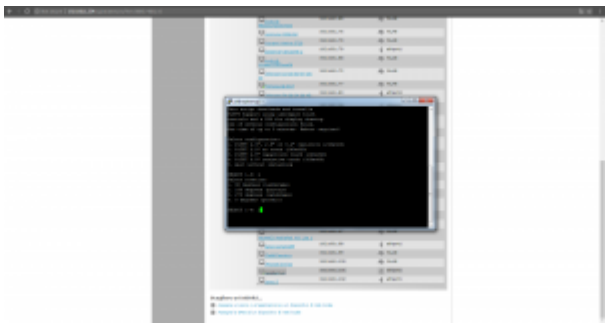
Dopo avere collegato il proprio Raspberry all'alimentazione (e eventualmente alla rete, se si sta usando l'ethernet), si può accedere al terminale remoto conoscendo il suo indirizzo IP. L'indirizzo Il programma che simula un terminale remoto SSH su Windows si chiama Putty, basta indicare l'indirizzo e premere **Open**. Quando viene richiesto il login e la password, basta indicare rispettivamente **pi** e **raspberry**. Per chi non lo sapesse, la password non compare mentre le si scrive, come da tradizione dei sistemi Unix.



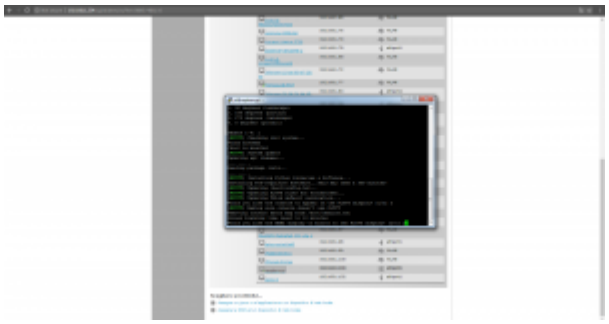
Se il login è corretto si accede al terminale del Raspberry:

Per configurare il TFT di Adafruit bisogna dare i seguenti comandi:

Quando la configurazione inizia, vengono richieste due informazioni: una sul modello di schermo che si sta usando (nell'esempio il numero 4, quello da 3.5 pollici), e una sull'orientamento con cui è fissato sul Raspberry (tipicamente la 1, classico formato orizzontale).



Alla fine dell'installazione dei vari software necessari, viene anche richiesto come usare lo schermo: alla prima richiesta, quella sulla console, conviene rispondere **n**, mentre alla seconda (quella sul mirroring HDMI) si deve rispondere **y**. In questo modo sullo schermo touch verrà presentata la stessa immagine che si può vedere dall'uscita HDMI.



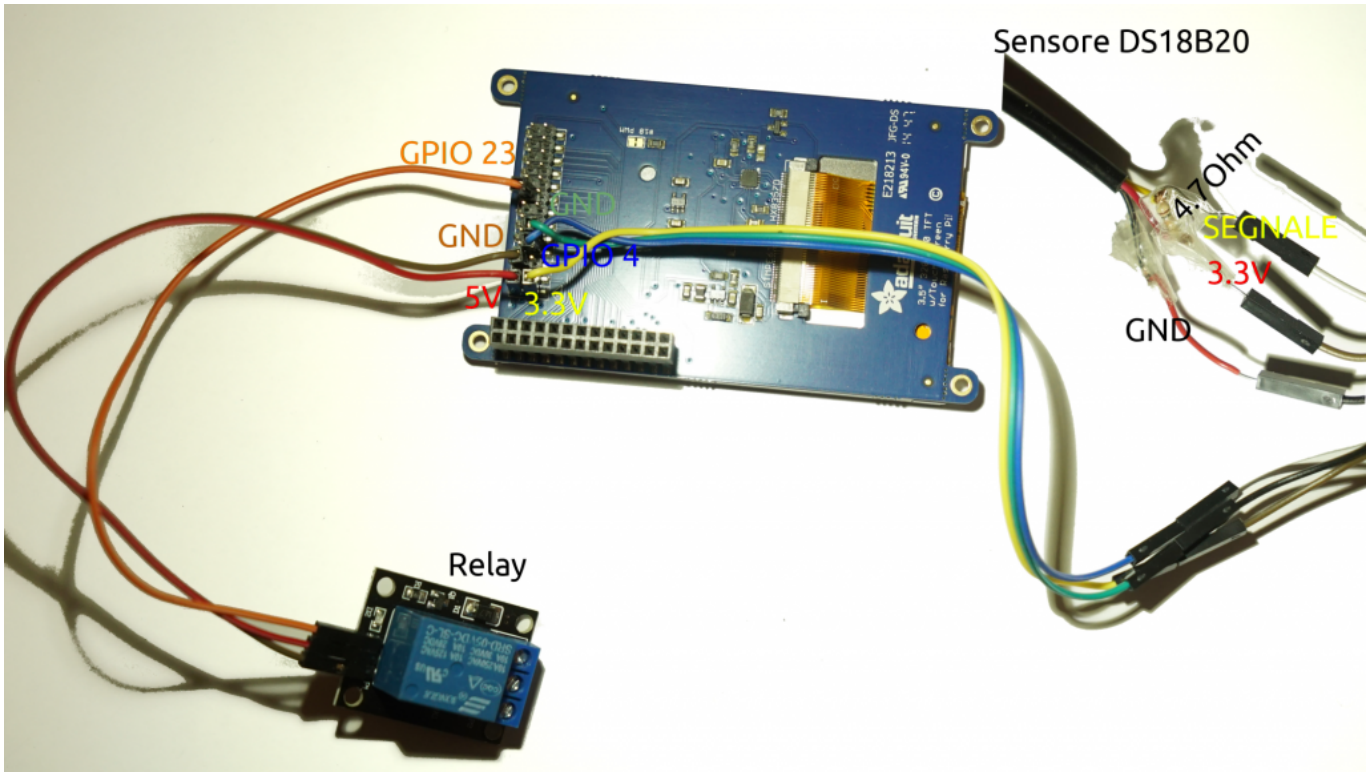
Un appunto (temporaneo) su Raspbian Buster
Se state usando l'immagine che abbiamo fornito all'inizio dell'articolo, basata su Raspbian Stretch, vi sarete accorti che lo script di Adafruit non funziona. Questo perché al momento il pacchetto tslib, necessario per lo script, non è disponibile per Buster, visto che si tratta di una versione non stabile. Per aggirare il problema, si può utilizzare

questa versione modificata dello script: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/adafruit-pitft.sh. Basta scaricarlo con il comando

Le altre istruzioni non cambiano. In futuro, quando Raspbian Buster verrà rilasciato ufficialmente, non sarà necessario usare questa versione modificata e si potrà fare riferimento all'originale.

I collegamenti del relè e del termometro

Quando si monta lo schermo sul Rasberry, i vari pin della scheda vengono coperti. È tuttavia possibile continuare a usarli perché lo schermo ce li ha doppi. Il sensore di temperatura e il relay potranno quindi essere collegati direttamente ai pin maschi che si trovano sullo schermo, seguendo lo stesso ordine dei pin sul Raspberry.



Ovviamente, un Raspberry offre molti pin, per collegare sensori e dispositivi, ma bisogna stare attenti a non usare lo stesso pin per due cose diverse. Per esempio, se stiamo usando il TFT da 3.5 pollici, possiamo verificare sul sito pinout che i contatti che usa sono il **18,24,25,7,8,9,10,11**. Rimane quindi perfettamente libero sul lato da 5Volt il pin 23, mentre sul lato a 3Volt è libero il pin 4. Il primo verrà usato come segnale di output per il relay, mentre il secondo come segnale di input del termometro. Il relay va collegato anche al pin 5V e GND, mentre il sensore va collegato al pin 3V e al GND.

La libreria per accedere ai pin GPIO dovrebbe già essere presente su Raspbian, mentre per installare quella relativa al sensore di temperatura si può dare il comando

Bisogna anche abilitare il modulo nel sistema operativo:

Dopo il riavvio diventa possibile utilizzare la libreria w1 per l'accesso al sensore di temperatura.

Un codice di test dal terminale

Possiamo verificare il funzionamento del relay e del termometro con un programma molto semplice da eseguire sul terminale:

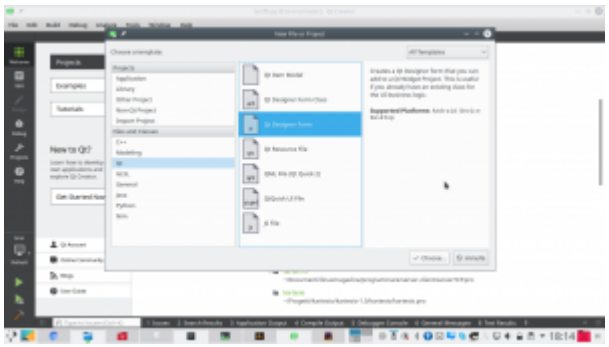
Per provare il programma basta dare i seguenti comandi:

Che cosa fa questo programma? Prima di tutto si assicura che siano caricati i moduli necessari per accedere ai pin GPIO e al sensore di temperatura. Poi esegue un ciclo infinito accendendo il relay, leggendo la temperatura attuale, aspettando 5 secondi, spegnendo il relay, leggendo ancora la

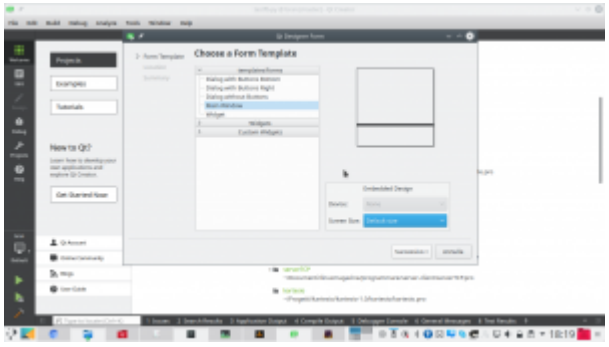
temperatura, e attendendo altri 5 secondi. Per terminare il programma, basta premere **Ctrl+C**.

L’interfaccia grafica

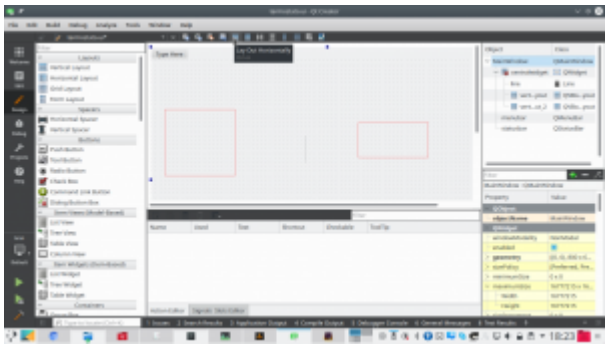
L’interfaccia grafica del nostro termostato è disponibile, assieme al resto del codice, nel repository Git: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/termostato.ui. Tuttavia, per chi volesse disegnarla da se, ecco i passi fondamentali. Prima di tutto si apre l’IDE QtCreator, creando un nuovo file di tipo Qt Designer Form.



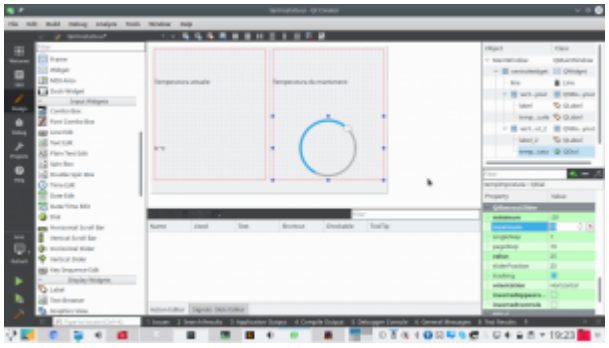
Il template da utilizzare è Main Window, perché quella che andiamo a realizzare è la classica finestra principale di un programma. Per il resto, la procedura guidata chiede solo dove salvare il file che verrà creato.



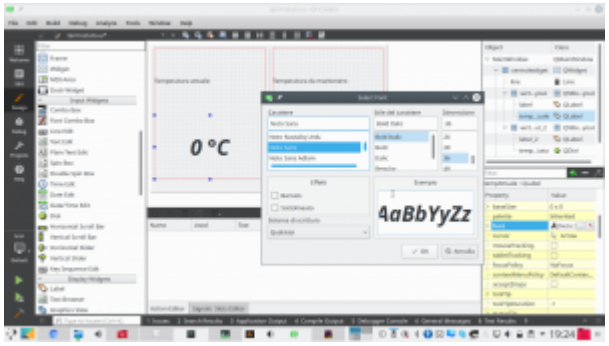
Quando si disegna una finestra, la prima cosa da fare è dividere il contenuto in layout. Considerando il nostro progetto, possiamo aggiungere due oggetti **Vertical Layout**, affiancandoli. Poi, con la barra degli strumenti in alto, impostiamo il form con un **Layout Horizontally**. I due layout verticali sono ora dei contenitori in cui possiamo cominciare ad aggiungere oggetti.



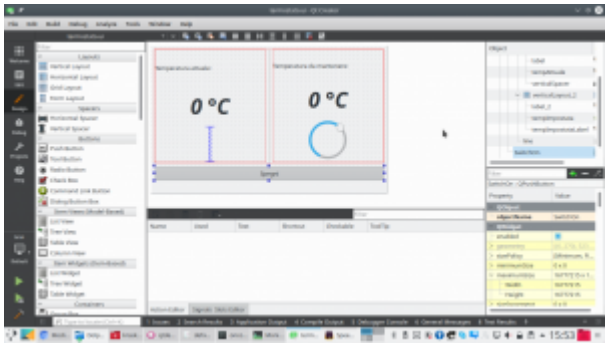
Inseriamo un paio di **label**: in particolare, una dovrà essere chiamata **tempAttuale**, e un'altra **tempImpostataLabel**. Queste etichette conterranno rispettivamente il valore della temperatura attualmente registrata dal termometro e quello che si è deciso di raggiungere (cioè la temperatura da termostatare). Nello stesso layout di **tempImpostataLabel** inseriamo un oggetto **Dial**, che chiamiamo **tempImpostata**. Si tratta di una classica rotella, proprio come quelle normalmente presenti sui termostati fisici. Tra le proprietà di questa dial, indichiamo i valori che desideriamo come minimo (**minimum**), massimo (**maximum**), e predefinito **(value).** Poi possiamo cliccare col tasto destro sulla **menubar** del form e scegliere di rimuoverla, così da lasciare spazio ai vari oggetti dell'interfaccia, tanto non useremo i menù.



Cliccando col tasto destro sui vari oggetti, è possibile personalizzarne l'aspetto. Per esempio, può essere una buona idea dare alle etichette che conterranno le due temperature una formattazione del testo facilmente riconoscibile, con una dimensione del carattere molto grande.



Infine, si può aggiungere, dove si preferisce ma sempre usando dei layout, un **Push Button** chiamato **SwitchOn**. Questo pulsante servirà per spegnere il termostato manualmente, in modo che il relay venga disattivato a prescindere dalla temperatura. Questa è una buona idea, perché se si sta via da casa per molto tempo non ha senso che il termostato scatti per tenere la casa riscaldata sprecando energia. Il pulsante che abbiamo inserito deve avere le proprietà **checkable** e **checked** attivate (basta cliccare sulla spunta), in modo da farlo funzionare non come un pulsante ma come un interruttore, che mantiene il proprio stato acceso/spento.



L'interfaccia è ora pronta, tutti i componenti fondamentali sono presenti. Per il resto, è sempre possibile personalizzarla aggiungendo altri oggetti o ridisegnando i layout.

Il codice del termostato

Cominciamo ora a scrivere il codice Python che permetterà il funzionamento del termostato:

All'inizio del file si inserisce la classica shebang, il

cancelletto con il punto esclamativo, per indicare l'interprete da usare per avviare automaticamente questo script Python3 (che non va quindi confuso col vecchio Python2). Si indica anche la codifica del file come utf-8, utile se si vogliono usare lettere accentate nei testi. Poi, si importano le varie librerie che abbiamo già visto essere utili per accedere ai pin GPIO del Raspberry, al termometro, e alle funzioni di sistema per misurare il tempo.

Ora dobbiamo aggiungere le librerie PySide2, in particolare quella per la creazione di una applicazione (**QApplication**). In teoria potremmo farlo con una sola riga di codice. In realtà, è preferibile usare questo sistema di blocchi try-except, perché lo utilizziamo per installare automaticamente la libreria usando il sistema di pacchetti **pip**. In poche parole, prima di tutto si prova (try) a importare la libreria **QApplication**. Se non è possibile (except), significa che la libreria non è installata, quindi si utilizza l'apposita funzione di pip per installare automaticamente la libreria. E siccome ci vorrà del tempo è bene far apparire un messaggio che avvisi l'utente di aspettare. È necessario usare due formule differenti perché, anche se al momento nella versione 3.6 di Python il primo codice funziona, in futuro sarà necessario utilizzare la seconda forma. Visto che la transizione potrebbe richiedere ancora qualche anno, con sistemi che usano ancora la versione 3.6 e altri con la 3.7, è bene avere entrambe le opzioni. Il vero vantaggio di questo approccio è che se si sta realizzando un programma realizzato con alcune librerie non è necessario preoccuparsi di distribuirle col programma: basta lasciare che sia Python stesso a installarla al primo avvio del programma. L'installazione è comunque possibile solo per le piattaforme supportate dai rilasci ufficiali su pip della libreria, e nel caso di Raspbian conviene sempre installare le librerie a parte.

Se l’importazione della libreria QApplication è andata a buon fine, significa che PySide2 è installato correttamente. Quindi possiamo importare tutte le altre librerie di PySide che ci torneranno utili nel programma.

Definiamo una variabile globale da usare per tutte le prossime classi: questa variabile, chiamata **toSleep**, contiene l’intervallo (in secondi) ogni cui controllare la temperatura.

Per prima cosa creiamo una classe di tipo QThread. Ovviamente, i programmi Python vengono sempre eseguiti in un unico thread, quindi se il processore è impegnato a svolgere una serie di calcoli e operazioni in background (come il raggiungimento della temperatura) non può occuparsi anche dell’interfaccia grafica. Il risultato è che l’interfaccia risulterebbe bloccata, un effetto sgradevole per l’utente e poco pratico. La soluzione consiste nel dividere le operazioni in più thread: il thread principale sarà sempre assegnato all’interfaccia grafica, e creeremo dei thread a parte che possano occuparsi delle altre operazioni. Creare dei thread in Python non è semplicissimo, ma per fortuna usando i QThread diventa molto facile. Il QThread che stiamo preparando ora si chiama TurnOn, e servirà per accendere e spegnere il relay in modo da raggiungere la temperatura impostata. All’inizio della classe si definisce un **segnale** con valore booleano (**True** oppure **False**) chiamato **TempReached**. Potremo emettere questo segnale per far sapere al thread principale (quello dell’interfaccia grafica) che la temperatura fissata è stata raggiunta e il termostato ha fatto il suo lavoro. Nella funzione che costruisce il thread (cioè **__init__**), ci sono le istruzioni necessarie per accedere ai pin GPIO del relay e e al sensore. Il pin cui è collegato il relay viene memorizzato nella variabile **self.relayPin**, mentre il sensore sarà raggiungibile tramite l’oggetto **self.sensor**. La funzione prende in argomento **w**, un oggetto che rappresenta la finestra del programma (che passeremo al thread dell’interfaccia

grafica stessa). In questo modo le funzione del thread potranno accedere in tempo reale all'interfaccia e interagire.

La funzione **run** viene eseguita automaticamente quando avviamo il thread: potremmo inserire le varie operazioni al suo interno, ma per tenere il codice pulito ci limitiamo a usarla per chiamare a sua volta la funzione **reachTemp**. Sarà questa a svolgere le operazioni vere e proprie. Prima di vederla, definiamo un'altra funzione: **readTemp**. Questa funzione non fa altro che leggere la temperatura attuale, scriverla sul terminale per debug, e restituirla. Ci tornerà utile per leggere facilmente la temperatura e controllare se è stata raggiunta quella prefissata. Nella funzione **reachTemp** per prima cosa si dichiara di avere bisogno della variabile globale **toSleep**. Poi si inserisce un blocco try-except: in questo modo, se per qualche motivo l'attivazione del relay dovesse fallire, verrà lanciato il segnale **TempReached** con valore **False** e nell'interfaccia grafica potremo tenerne conto per avvisare l'utente che qualcosa è andato storto. Se invece va tutto bene, si inizia un ciclo while, che rimarrà attivo finché il pulsante presente nell'interfaccia grafica (che abbiamo chiamato SwitchOn) è premuto (cioè nello stato **checked**). Ciò significa che appena l'utente cliccherà sul pulsante per farlo passare allo stato "spento" (cioè "non checked") il ciclo si fermerà. Nel ciclo, si controlla se la temperatura attuale (ottenuta grazie alla funzione **readTemp**) sia inferiore alla temperatura impostata per il termostato. In caso positivo bisogna assicurarsi che il relay sia acceso, quindi il suo pin si imposta con valore **HIGH**. E poi si attende il tempo prefissato prima di fare un altro ciclo e quindi controllare di nuovo la temperatura. Se, invece, la temperatura attuale è maggiore di quella da raggiungere, vuol dire che possiamo spegnere il relay impostando il suo pin al valore **LOW**, ed emettere il segnale **TempReached** col valore **True**, visto che è andato tutto bene. Abbiamo quindi un ciclo che si ripete, per esempio, ogni 10 secondi finché viene

raggiunta la temperatura impostata, e a quel punto si interrompe.

Poi creiamo un'altra classe di tipo QThread, stavolta chiamata **ShowTemp**. In questa classe non facciamo altro che leggere la temperatura attuale, dal sensore, e inserirla nella **label** che abbiamo dedicato proprio a presentare questo valore all'utente. Avremmo potuto inserire questa funzione nello stesso QThread già creato per regolare la temperatura, visto che poi lo possiamo lanciare più volte e con argomenti diversi. Quindi avremmo potuto usare un QThread unico con le due funzioni da attivare separatamente. Tuttavia, quando si fanno operazioni diverse è una buona idea tenere il codice pulito e creare QThread separati. Esattamente come per il QThread precedente, la funzione di costruzione della classe (la solita **__init__**) richiede come argomento **w**, l'oggetto che rappresenta la finestra dell'interfaccia grafica. Viene anche creato l'oggetto **sensor**, per accedere al sensore di temperatura. Anche in questa classe inseriamo la funzione **readTemp**, uguale a quella del QThread precedente, e per semplificare stavolta scriviamo le istruzioni direttamente nella funzione **run**. Anche in questo caso si accede alla variabile globale **toSleep**. Il codice che svolge davvero le operazioni è un semplice ciclo infinito (**while True**) che imposta un nuovo valore come testo della label presente nell'interfaccia grafica (cioè **self.w**). L'etichetta in questione si chiama **tempAttuale**, e possiamo assegnarle un testo usando la funzione **setText**. Il testo viene creato prendendo la temperatura (ottenuta dalla funzione **readTemp** e traducendo il numero in una stringa di testo) e aggiungendo il simbolo dei gradi Celsius. Poi, si aspetta il tempo preventivato prima di procedere a ripetere il ciclo.

Il risultato che otterremo, con l'interfaccia grafica interattiva

La classe **MainWidow**, di tipo QMainWindow, conterrà il codice necessario a far apparire e funzionare la nostra interfaccia grafica.

La prima cosa da fare, nella funzione che costruisce la classe (la solita **__init__**) è caricare, in lettura (**QFile.ReadOnly**) il file che contiene l'interfaccia grafica stessa. Il file in questione si trova nella stessa cartella dello script attuale, e si chiama **termostato.ui**, quindi possiamo scoprire il suo percorso completo estraendo dal nome dello script (**sys.argv[0]**) la cartella in cui si trova (con la funzione **os.path.dirname**) e risalire al percorso assoluto con la funzione **os.path.abspath**. L'interfaccia grafica viene interpretata con la libreria **QUiLoader**, e memorizzata nell'oggetto **self.w**. Da questo momento sarà quindi possibile accedere ai vari componenti dell'interfaccia grafica usando questo oggetto. Affinché l'interfaccia grafica venga utilizzata per la finestra che stiamo costruendo, bisogna impostare l'oggetto **self.w** come **CentralWidget**. Possiamo impostare un titolo per la finestra con la funzione **self.setWindowTitle**, tipica di ogni QMainWindow. Ora possiamo cominciare a rendere interattiva l'interfaccia grafica: per farlo dobbiamo collegare i segnali degli oggetti dell'interfaccia alle funzioni che si occuperanno di gestirli. Per esempio, dobbiamo collegare il segnale **clicked** del pulsante **SwitchOn** a una funzione che chiameremo **self.StopThis**. Lo facciamo usando la funzione **connect** del segnale di questo

pulsante. La scrittura è molto semplice: si tratta semplicemente di un sistema a scatole cinesi. Per esempio, anche quando colleghiamo il movimento della rotella (la QDial per impostare la temperatura) alla funzione **setTempImp** non facciamo altro che prendere l'oggetto che rappresenta l'interfaccia grafica, cioè **self.w**, e puntare sulla QDial al suo interno, che avevamo chiamato **tempImpostata**. All'interno di questo oggetto, andiamo a recuperare il segnale **valueChanged**, che viene emesso dalla QDial stessa nel momento in cui l'utente modifica il suo valore spostando la rotella, e per questo segnale chiamiamo la funzione **connect**. Alla funzione bisogna soltanto assegnare il nome (completo di **self.**, non dimentichiamo che è un membro della classe Python che stiamo scrivendo) della funzione che dovrà essere chiamata. Va indicato solo il nome, senza le parentesi, quindi si scrive **self.setTempImp** e non **self.setTempImp()**.

Sempre nella funzione **__init__** si provvede a dare i comandi necessari per attivare i moduli di sistema che forniscono il controllo del sensore e dei pin GPIO. Poi impostiamo la variabile **self.alreadyOn** a False: si tratta di un semplice flag che useremo per capire se il relay sia già stato attivato, o se sia necessario attivarlo, quindi ovviamente all'avvio del programma è False perché il relay è ancora spento. Ora si può accedere alla QDial e impostare manualmente il suo valore iniziale, con la funzione **setValue**, per esempio a 25 gradi. Poi va chiamata manualmente la funzione **setTempImp**, con lo stesso valore in argomento, per essere sicuri che il programma controlli se sia necessario accendere o spegnere il relay per raggiungere la temperatura in questione. La variabile **self.stoponreached** verrà utilizzata soltanto per decidere se disattivare il termostato una volta raggiunta la temperatura: di norma non è necessario, anzi, si preferisce che il termostato rimanga vigile per riaccendere il relay qualora la temperatura dovesse scendere nuovamente. Ma per funzioni di test o casi di abitazioni con un isolamento

davvero buono può avere senso impostare questa variabile a True. L'ultima cosa da fare prima di concludere la funzione di costruzione dell'interfaccia grafica è creare il thread che si occupa di leggere la temepratura attuale dal sensore e scriverla nell'apposita label. Basta creare un oggetto di tipo **ShowTemp**, perché questo è il nome che abbiamo scelto per la classe di questo QThread, indicando l'oggetto **self.w** come argomento, così le funzioni del thread potranno accedere all'interfaccia grafica di questa finestra. Il thread viene avviato usando la funzione **start**. È importante non confondersi: quando si scrive la classe del QThread il codice va messo nella funzione **run**, ma quando lo si avvia si chiama la funzione **start**, perché così vengono eseguiti una serie di controlli prima dell'effettivo inizio delle operazioni.

Definiamo due funzioni: una è **reached**, e l'altra **itIsOff**. La funzione **reached** verrà chiamata automaticamente quando la temperatura impostata per il termostato viene raggiunta. A questo punto possiamo decidere cosa fare: se la variabile **stoponreached** è impostata a True, chiameremo la funzione **itIsOff**, così da disattivare il termostato. In caso contrario, non è necessario fare nulla, ma volendo si potrebbe modificare l'aspetto dell'interfaccia grafica (per esempio colorando di verde l'etichetta con la temperatura) per segnalare che la temperatura è stata raggiunta. La funzione **itIsOff**, come abbiamo già suggerito, si occupa di disattivare il termostato. Per farlo, imposta come False il pulsante presente nell'interfaccia grafica: siccome si comporta come un interruttore, se il suo stato **checked** è falso il pulsante è disattivato. E, come avevamo visto nella classe del thread TurnOn, il ciclo che si occupa di controllare se sia necessario tenere il relay rimane attivo solo se il pulsante è **checked**. Poi viene chiamata la funzione StopThis, che si occupa di modificare il pulsante (che da "Spegni" deve diventare "Accendi").

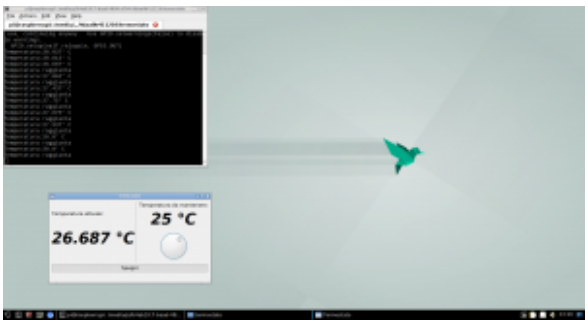
La funzione **dostuff** è quella che si occupa effettivamente di creare il thread dedicato al controllo del relay. Prima di tutto, si controlla che il thread non sia già stato avviato, usando il flag **alreadyOn** che avevamo creato all'inizio del codice di questa classe. Se il thread non è già attivo, lo si crea passandogli l'oggetto **self.w** così da permettere al thread l'accesso all'interfaccia grafica. Poi colleghiamo il segnale **TempReached**, che avevamo creato per il thread **TurnOn**, alla funzione **self.reached**. Si collega anche il segnale **finished**, dello stesso thread, alla funzione **itItOff**, così se per un motivo o l'altro il thread dovesse terminare la funzione adeguerebbe lo stato del pulsante presente nell'interfaccia grafica. Alla fine, si avvia il thread e si imposta il flag **alreadyOn** come True.

La funzione **StopThis** controlla lo stato attuale del pulsante: se è attivato, il suo testo deve essere impostato a "Spegni", e bisogna ovviamente chiamare la funzione **dostuff** in modo che venga lanciato il thread, se necessario. Viceversa, se il pulsante è disattivato, il suo testo deve essere "Accendi", così l'utente capirà che premendo il pulsante può attivare il sistema, e il flag **alreadyOn** va impostato a False, per segnalare che al momento il thread è disattivato (ricordiamo che se il pulsante è disattivato, anche il thread **TurnOn**, inserito in questa finestra col nome **self.myThread**, si disattiva automaticamente).

L'ultima funzione che inseriamo nella classe della finestra è **setTempImp**, ed è la funzione che abbiamo collegato al segnale **valueChanged** della QDial. Quando l'utente sposta la rotella, verrà chiamata questa funzione. Si può notare che questa funzione è leggermente diversa dalle altre che abbiamo scritto finora per reagire ai segnali dell'interfaccia grafica: ha un argomento, chiamato **arg1**. Questo perché il segnale **valueChanged** offre alla funzione chiamata il valore attuale della QDial. Nel nostro caso, quindi, **arg1** contiene la

temperatura che l’utente vuole impostare, quindi possiamo direttamente inserirla nell’etichetta **tempImpostataLabel**, che abbiamo creato nell’interfaccia grafica proprio per presentare la temperatura selezionata. Poi, per sicurezza, chiamiamo la funzione **dostuff** in modo da attivare il thread che fa funzionare il relay se è necessario.

Terminate le classi, possiamo scrivere il codice principale del programma. Per prima cosa, il programma crea una **QApplication**, necessaria per le librerie Qt. Poi possiamo creare una istanza della finestra principale, memorizzandola nell’oggetto **w**. Ora possiamo impostare alcune caratteristiche della finestra. Per esempio, la dimensione: se stiamo usando lo schermo PiTFT3.5, la risoluzione ideale è 640×480: naturalmente, si possono modificare i parametri sulla base delle proprie esigenze. Infine, si fa apparire la finestra con la funzione **show**. E quando questa verrà chiusa (cosa che nel nostro caso non dovrebbe succedere, ma è un buona idea tenere in considerazione l’ipotesi), si chiude normalmente il programma, fornendo alla riga di comando il risultato dell’esecuzione dell’applicazione (quindi eventuali codici d’errore se qualcosa dovesse non funzionare correttamente).



L’applicazione eseguita dal desktop del Raspberry, per provare il suo funzionamento

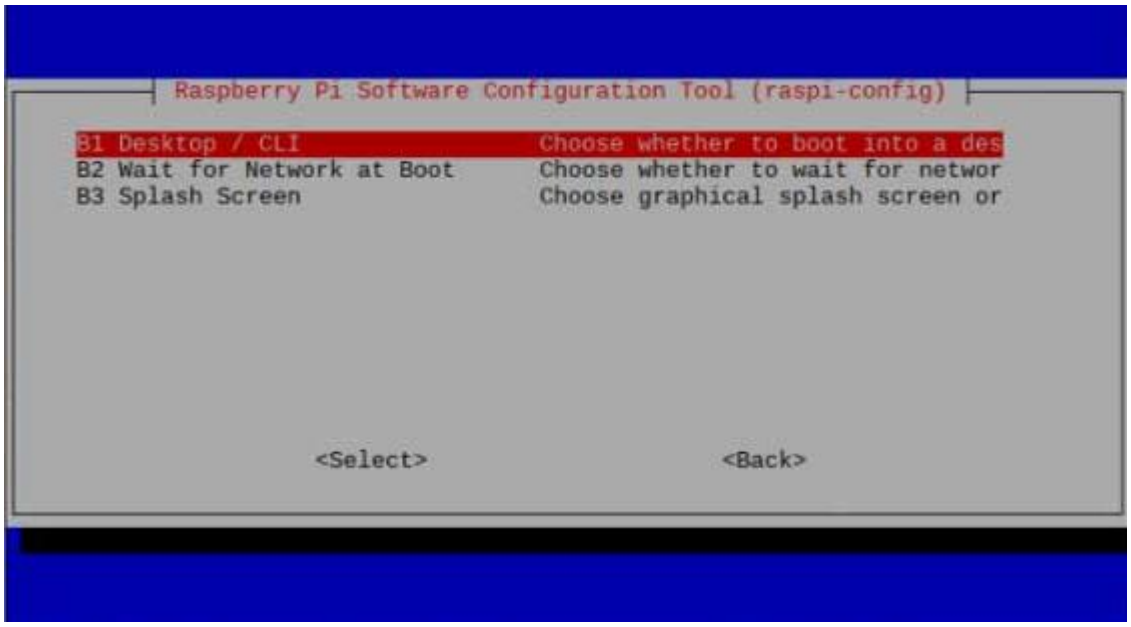
Avvio automatico

Siccome la nostra applicazione è pensata per apparire sullo schermo all'avvio del sistema, dobbiamo preparare un piccolo script per automatizzare la sua installazione:

Prenderemo come riferimento l'utente pi, che è sempre disponibile su Raspbian. Con queste prime righe di codice creiamo un servizio di sistema che esegue il login automatico per l'utente pi sul terminale tty1. Così, all'avvio del sistema non sarà necessario digitare la password, si avrà subito un terminale funzionante.

Si modificano due file di configurazione: con la modifica a xinit aggiungiamo il comando cat all'avvio del server grafico: questo permette di tenerlo in stallo e evitare eventuali procedure automatiche. La modifica a bashrc ci permette di fare un controllo appena viene aperto un terminale per l'utente **pi**. Se il nome del terminale è tty1 (su GNU/Linux ci sono diversi terminali disponibili) lanciamo sia lo script di avvio del nostro programma, sia il server grafico Xorg. Aver fatto questo controllo è importante, perché così il programma termostato verrà avviato automaticamente solo sul terminale tty1, quello che ha l'autologin, e non su tutti gli altri.

Infine, si scrive lo script di avvio del programma termostato: inizialmente, lo script attende un secondo, in modo da essere sicuro che il server grafico sia pronto a funzionare. Poi, lancia Python3 con il nostro programma.



C’è un dettaglio: con l’immagine di Raspbian Buster fornita all’inizio dell’articolo l’autologin potrebbe non funzionare correttamente, e questo perché Raspbian usa carica una immagine di splash per il boot che impedisce il corretto avvio del server grafico per come lo abbiamo configurato. La soluzione è disabilitare il boot con splash grafica, visto che comunque non ci serve, usando il comando **sudo raspi-config** nella sezione **Boot**, selezionando l’opzione **Splash Screen** e impostandola come disabilitata.

Il codice completo
Potete trovare il codice completo nel repository git https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/
Per scaricarlo potete dare il comando

da un terminale GNU/Linux come quello del Raspberry, oppure usare le varie interfacce grafiche di Git disponibili. O, anche, scaricare i file singolarmente dalla pagina https://codice-sorgente.it/cgit/termostato-raspberry.git/plain/. Nel repository è presente un README con i comandi da eseguire per installare correttamente il programma sulla

versione di **Raspbian Buster** che proponiamo. C’è da dire che il codice che abbiamo presentato è pensato per un uso accademico, non professionale: una applicazione per un termostato può essere più semplice, il codice è prolisso in diversi punti per facilitare la comprensione a chi non sia pratico delle librerie Qt.