PhotoFish
DIGITAL FRAME WIFI

Versione connessa delle popolari cornici digitali, che permette di riprodurre contenuti multimediali ricevuti in wireless grazie a Fishino e allo shield TFT. L'occasione ci permette di fare didattica sulla gestione delle immagini. Prima puntata.

di MASSIMO DEL FEDELE

Conoscerete probabilmente le "cornici digitali", meglio note come "digital frame", ossia quei dispositivi multimediali che sembrano un quadretto e che possono riprodurre una serie di fotografie presenti su un supporto come SD-Card o Pen Drive USB in varie modalità: ad esempio la sequenza ciclica con eventuali effetti video. Questi dispositivi, che nelle versioni più avanzate possono riprodurre anche filmati, sono limitate dal fatto che possono attingere solamente a un supporto di memoria permanente locale. Partendo da questa considerazione e guardandoci intorno, abbiamo voluto

creare qualcosa di simile, però svincolato dai limiti della memoria locale; come scoprirete, il progetto descritto in queste pagine è sostanzialmente una digital frame con una marcia in più, perché capace di caricare e visualizzare contenuti attraverso la connettività WiFi fornita dalle nostre schede Fishino e mostrarli sfruttando le potenzialità dei nuovi modelli a 32 bit.
Iniziamo con le caratteristiche della nostra cornice, che possiamo definire "telematica":

- possibilità di inviare le immagini tramite i servizi del "social" Telegram;
- possibilità (utilizzando Fishino32) di riprodurre file audio, sempre inviati da Telegram (estensione futura);
- ridimensionamento e rotazione automatiche delle immagini per ottimizzarle in base alla risoluzione del display;
- appena 2 secondi richiesti per la visualizzazione di un contenuto a decorrere dall'invio.

Tale "ritardo" si ottiene utilizzando per il progetto una Fishino a 32 bit; se si adotta una Mega ad 8 bit, il tempo aumenta a circa 30 secondi.
Per quanto il progetto possa fare di più, in questa prima fase abbiamo appositamente evitato di introdurre alcune funzionalità, che probabilmente aggiungeremo in futuro; tra esse, la possibilità di memorizzare su SD le immagini ricevute e farle scorrere secondo criteri predeterminati e/o impostabili tramite messaggi Telegram.
Allo stesso modo, è possibile implementare un menu sullo schermo del display TFT (che è un touch resistivo) in modo da poter configurare l'applicazione tramite un menu su schermo.
Lo spazio per queste successive aggiunte non manca, specialmente sul Fishino32, quindi nei prossimi mesi vedremo di aggiungerle in modo da estendere le funzionalità, creando nuove librerie software che saranno comunque utilizzabili anche per altri scopi.
Abituati a utilizzare i nostri smartphone per visualizzare immagini e video in tempo reale, queste caratteristiche, impensabili anche solo una decina di anni fa, non stupiscono più di tanto, se non fosse che nei moderni cellulari e Personal Computer sono contenuti microprocessori velocissimi e multi-core, con i quali anche programmi poco ottimizzati sono in grado di fare cose notevoli; sono presenti inoltre quantità enormi (gigabyte) di memoria RAM in cui parcheggiare le immagini e i dati ricevuti, per poi elaborarli utilizzando altre quantità non trascurabili di risorse. Quindi, perché non utilizzarli? Ebbene, i motivi sono tanti, ma si possono riassumere in questi tre:

- costi;
- consumi di energia;
- necessità di un complesso sistema operativo.

Gli smartphone o, peggio, i PC, consumano decine se non centinaia di watt, costano decisamente più di un microcontroller (escludendo forse la fortunata famiglia della Raspberry Pi, che ci si avvicina pur dovendo mettere in computo il costo delle periferiche) e, cosa non trascurabile, richiedono tempi di avvio non brevi. Un microcontroller, per contro, "parte" non appena viene alimentato, consuma e costa poco, quindi può tranquillamente venire dedicato ad uno scopo specifico come quello che ci prefiggiamo.
Dove sta l'inghippo, quindi? Principalmente, appunto, nella scarsità di risorse disponibili nei microcontrollori.

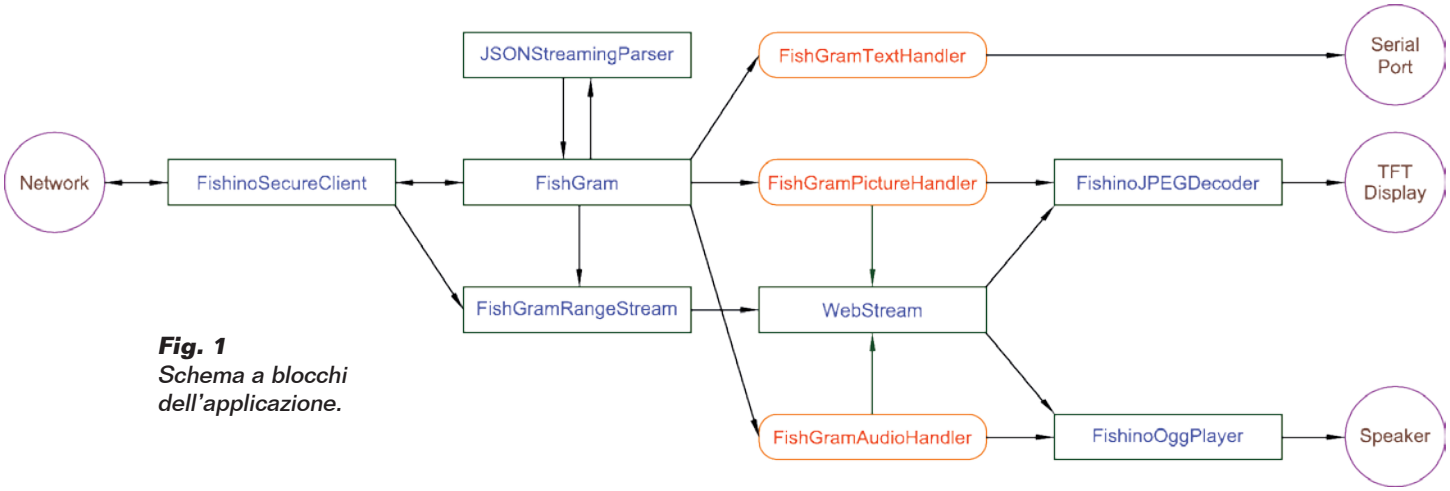
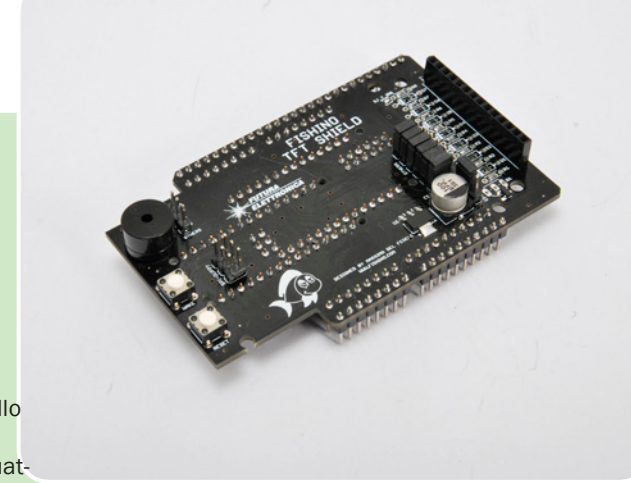


Fig. 1
Schema a blocchi dell'applicazione.

Applicare il display con **TFT SHIELD**



Per realizzare il digital Frame WiFi utilizziamo il TFT shield descritto nell'articolo ad esso dedicato nel fascicolo di giugno scorso, che monta a "sandwich" il display TFT. Lo shield è adatto ad essere montato, con gli adattamenti dei connettori descritti nello stesso articolo, a tutte le board Fishino e può ospitare display grafici che richiedano il minor numero di I/O possibili; in pratica supporta display con comunicazione SPI, facilmente reperibili nei formati 2,4 o 2,8 pollici. Questo tipo di display utilizza le linee SPI (MISO, MOSI e SCK), in comune con il modulo WiFi e le schede SD, una linea di selezione ed una di controllo per il display ed altre 2 linee per lo schermo touch-sensitive, quindi quattro I/O utilizzati in "esclusiva" contro 10÷12 minimo per i modelli ad interfaccia parallela. Se volete utilizzare un display senza touch-panel (è possibile farlo, almeno in questa applicazione) risparmiate due linee di I/O.
Per la gestione dello shield abbiamo approntato tre librerie specifiche che ne consentono il controllo completo da parte di Arduino e Fishino. Queste librerie sono:

- FishinoGFX, versione praticamente identica all'analoga di Adafruit, che gestisce le funzioni grafiche "ad alto livello";
- FishinoILI9341, che gestisce le funzioni di interfaccia con il display a livello hardware; anche questa libreria è stata realizzata partendo dall'analoga di Adafruit, ma con modifiche abbastanza sostanziali;
- FishinoXPT2046; che gestisce il touch screen, scritta da zero di nostro pugno.

A differenza di un programma per Personal Computer, uno sketch capace di realizzare l'applicazione che ci proponiamo richiede consistenti ottimizzazioni sia per funzionare a velocità discrete, sia per occupare il minimo indispensabile di memoria RAM, sempre scarsa in questo tipo di dispositivi: si parla di kilobyte e non di megabyte o gigabyte. Anche una Fishino32, che con i suoi 128 k di RAM è un gigante tra i microcontroller, se paragonata a un sistema a microprocessore con qualche gigabyte di RAM a disposizione, diventa una "formichina".
Per capirne le problematiche, che verranno comunque approfondite in seguito nella descrizione del funzionamento, basti sapere che un'immagine, per quanto caratterizzata da una risoluzione grafica non particolarmente elevata (intorno ai 400x400 pixel con profondità di colore di 16 bit) occupa, non compressa, ben 320 kbyte di memoria: quasi il triplo di quanto disponibile su una board Fishino32 e ben 40 volte quella di una Arduino o Fishino Mega.
Se un'immagine non compressa, quindi visualizzabile direttamente, occupa quantità notevoli di memoria, nemmeno le versioni compresse scherzano (.jpg, formato JPEG): si viaggia sempre intorno ai 50÷100 kB ed oltre, quindi già di per se sufficienti a riempire tutta la RAM o quasi, anche sulla Fishino32.

COME FUNZIONA?

Dunque, come si fa con una scheda Arduino a realizzare l'applicazione del caso? Esclusa la possibilità di operare normalmente, ovvero di scaricare dalla rete l'immagine intera, decomprimerla in memoria, trasferirla al display, eccetera, che cosa ci resta? Semplicemente, occorre lavorare byte per byte, processando i dati man mano che vengono ricevuti e trasferendone direttamente il risultato al display; in questo caso ci serve un buffer limitato ma veloce.
L'applicazione che realizza tutto ciò è piuttosto complessa, ma ben scomponibile in elementi logici relativamente semplici da comprendere; iniziamo quindi dal classico schema a blocchi, mostrato nella **Fig. 1**. Descriveremo qui tutti gli elementi, approfondendo in dettaglio il decoder JPEG anche dal punto di vista teorico, che riteniamo piuttosto interessante.

NOTE PRELIMINARI

Come visto negli applicativi precedenti, l'interazione tra Fishino e Telegram avviene attraverso

Listato 1

```
_client << F("GET /bot");
if (_flashToken)
   _client << (const __FlashStringHelper *)_token;
else
   _client << _token;
_client << F("/getUpdates?offset=");
_client.print(id, DEC);
_client
   << F("&timeout=4&limit=1&allowed_updates=messages HTTP/1.1\r\n")
   << F("User-Agent: FishGram 1.0.0\r\n")
   << F("Host: api.telegram.org\r\n\r\n")
;
```

l'ausilio di un Bot, ovvero un "utente artificiale e virtuale" di Telegram, che potremo creare noi stessi sulla base delle nostre esigenze.
Il procedimento di creazione di un Bot è già stato ampiamente spiegato ad esempio nel fascicolo 208, quindi non lo ripeteremo; nel caso potete trovare le descrizioni dettagliate, sia nel predetto fascicolo, sia nel n° 212 di Elettronica In (o anche nell'articolo del "Termostato con Fishino" del fascicolo n° 215).
Quello di cui abbiamo bisogno da qui in poi è la cosiddetta TOKEN di accesso al bot, con la quale lo potremo contattare tramite richieste HTTPS.
L'uso dell'applicazione è semplicissimo, disponendo del nostro TFT shield: si inserisce il TFT shield nella scheda Fishino prescelta, quindi si monta il display nello shield; questo per l'hardware. Sul lato software, si modifica lo sketch aggiustando SSID e password della propria rete WiFi ed il Token del bot Telegram, che dovrete avere creato preventivamente come accennato.
Fatto questo, basta caricare lo sketch ed inviare al Bot un'immagine tramite Telegram: la vedrete apparire immediatamente sul display!
Prima di vedere il firmware che consente la visualizzazione dedichiamo qualche paragrafo alla descrizione dei vari moduli impiegati.

Listato 2

```
[
 ok:true,
 result:[
    {
         update_id:323513224,
         message:{
                  message_id:1164,
                  from:{
                           id:328328564,
                           first_name:"Massimo",
                           last_name:"Del Fedele"
                  },
                  chat:{
                           id:328328564,
                           first_name:"Massimo",
                           last_name:"Del Fedele",
                           type:"private"
                  },
                  date:1494343276,
                  photo:[
                           {
                                    file_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABCSeb-z9KYvV-U4BAAEC",
                                    file_size:1457,
                                    width:51,
                                    height:90
                           },
                           {
                                    file_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABCnm8_UrmclE-k4BAAEC",
                                    file_size:20309,
                                    width:180,
                                    height:320
                           },
                           {
                                    file_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABLFIAum3eG-X_E4BAAEC",
                                    file_size:91483,
                                    width:450,
                                    height:800
                           },
                           {
                                    file_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABHV37QcFKp9z-04BAAEC",
                                    file_size:162304,
                                    width:720,
                                    height:1280,
                           }
                  ]
         }
    }
 ]
]
```

MODULI FishGram E JSONStreamingParser

Entrambi i moduli sono già stati utilizzati in applicazioni precedenti, anche se in questo caso li abbiamo estesi (soprattutto il primo) per supportare gli elementi multimediali assenti nelle versioni precedenti, quando ci interessavano i soli messaggi di testo.

Iniziamo dalla lettura dei messaggi Telegram spiegando come funziona, fermo restando che pur avendolo visto abbastanza bene negli applicativi già realizzati, lo ripeteremo velocemente qui con le dovute aggiunte relative ad immagini e file audio.

Telegram ha due modalità di funzionamento: polled (che si potrebbe tradurre in "a richiesta") ed una specie di modalità event-driven (Webhook, letteralmente "aggancio al web"), nella quale un server si mette in ascolto ed il sistema di Telegram invia gli eventi non appena arrivano. La seconda modalità sembrerebbe allettante, se non fosse che Telegram è piuttosto esigente, richiedendo un server con certificati SSL, un nome di dominio, eccetera, rendendola impraticabile nel nostro caso, a meno di non appoggiarci ad un sistema esterno.

La modalità polled, per contro richiede l'invio periodico di richieste al sistema Telegram per verificare l'eventuale presenza di messaggi, che possono poi essere scaricati.

Invio "periodico" quanto? Qui occorre un compromesso; se facciamo richieste troppo frequenti, il sistema viene sovraccaricato e potrebbe decidere di "chiudere i rubinetti", impedendoci di ricevere aggiornamenti di stato.

Se, per contro, facciamo richieste troppo distanti nel tempo, pur non perdendo eventi (che vengono comunque memorizzati da Telegram per un periodo più che sufficiente) rischiamo di rispondere troppo lentamente ai nostri comandi.

Il compromesso che abbiamo trovato è di una richiesta ogni 5 secondi circa, quindi mediamente avremo una latenza di 2,5 secondi dal nostro comando alla sua esecuzione.

Vediamo ora come, una tipica richiesta a Telegram, può venire direttamente inserita nel campo URL del nostro browser; l'istruzione è:

https://api.telegram.org/bot<TOKEN>/getUpdates?offset=nnn&timeout=4&limit=1&allowed_updates=messages

Si tratta di una semplice richiesta HTTP di tipo GET; "<TOKEN>" rappresenta il "codice di accesso" che identifica il nostro bot, "getUpdates" è il comando di richiesta di aggiornamenti, "offset" è l'ID di messaggio a partire dal quale vogliamo richiedere gli aggiornamenti (solitamente fornito come l'id dell'ultimo messaggio ricevuto + 1, in modo da ricevere solo i successivi), "limit=1" indica che vogliamo ricevere un messaggio alla volta, "allowed_updates=messages" indica che vogliamo ricevere solo i messaggi e non, per esempio, gli ingressi/uscite dalla chat di altre persone.

Tutte le richieste al server di Telegram vanno fatte tramite il protocollo HTTPS, cosa che costituisce un grosso problema per molti shield WiFi che non lo supportano, ma non per la nostra scheda Fishino la quale, anche se con certi limiti, è in grado di stabilire connessioni sicure.

Ovviamente la nostra Fishino non dispone di un browser web ma di un client (per la precisione un **FishinoSecureClient**) tramite il quale possiamo inviare e ricevere dati tramite il protocollo TCP.

Lo spezzone di codice che esegue la richiesta è riportato nel **Listato 1**, nel quale, come potete notare, il codice fa più o meno quanto faremmo manualmente sulla riga di comando del browser, salvo aggiungere in testa il tipo di richiesta (GET) ed in coda degli elementi HTTP indispensabili, che sono il tipo di protocollo, HTTP/1.1, User-Agent ed Host.

Si noti il doppio "capolinea" (\r\n\r\n) che termina gli header HTTP ed avvia la richiesta.

In una richiesta di tipo GET è assente il corpo del messaggio (che in una richiesta di tipo POST, ad esempio, andrebbe messo dopo il doppio capolinea di cui sopra).

L'istruzione condizionale "if(_flashToken)" permette di supportare token sia nella memoria RAM che nella PROGMEM, cosa ottimale se il token è fisso e lo si vuol inserire direttamente nello sketch, in modo da risparmiare preziosa RAM nei processori in cui lo spazio della FLASH è separato da quello della RAM (architettura Harvard, come per esempio nel microcontrollore della scheda Fishino MEGA). Sulla Fishino32 l'area di indirizzamento è "piatta", quindi non occorre differenziare le due cose.

Una volta inviata la richiesta, Telegram risponde con un pacchetto HTTP contenente i soliti Header (che vengono saltati) ed un corpo in formato JSON; una risposta tipica è quella visibile nel **Listato 2**.

Chiaramente abbiamo inserito noi la formattazione, per rendere più chiaro il JSON ricevuto; nella realtà la risposta contiene lo stretto indispensabile, quindi niente capolinea, spazi non necessari, tabulazioni, ecc., come nello stesso JSON non formattato qui di seguito:

```
[ok:true,result:[{update_id:323513224,message:{message_
id:1164,from:{id:328328564,first_
name:"Massimo",last_name:"Del Fedele"},chat:{id:
328328564,first_name:"Massimo",last_name:"Del Fe
dele",type:"private"},date:1494343276,photo:[{fi
le_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABCSeb-
z9KYvV-U4BAAEC",file_size:1457,width:51,height:90},
{file_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABCnm8_
UrmclE-k4BAAEC",file_size:20309,width:180,height:320},{f
ile_id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABLFIAum3eG-
X_E4BAAEC",file_size:91483,width:450,height:800},{file_
id:"AgADBAADOKkxG1FikFALMHOINZs_th8fqRkABHV37QcFKp9z-
04BAAEC",file_size:162304,width:720,height:1280,}]}}]]
```

Non dimentichiamoci che la formattazione è utilissima ad un umano, ma controproducente per una macchina, che con essa si troverebbe a dover leggere un bel po' di caratteri non necessari.

Listato 3

```
const char *leggiStringa()
{
 static char buf[100];
 char *bufP = buf;
 while(client.available())
    *bufP++ = client.read();
 *bufP = 0;
 return buf;
}

void stampaValori(void)
{
 char nome[100];
 char valore[100];

 const char *stringa = leggiStringa();
 const char *p = stringa;

 // ciclo per ogni gruppo nome:valore
 // termina a fine stringa, quando trova un carattere nullo
 while(*p)
 {
    char *nomeP = nome;
    char *valoreP = valore;

    // ciclo di lettura nome
    while(*p && *p != ':')
         *nomeP++ = *p++;
    *nomeP = 0;

    // salta il ':'
    if(*p)
         p++;

    // ciclo di lettura valore
    while(*p && *p != ',')
         *valoreP++ = *p++;
    *valoreP = 0;

    // stampa nome e valore
    Serial << "NOME :" << nome << "\n";
    Serial << "VALORE:" << valore << "\n";

    // salta l'eventuale virgola
    if(*p)
         p++;
 }
}
```

Il JSON (formattato) è facilmente comprensibile, si tratta di una serie di informazioni nel formato **nome:valore**, che contengono tutto quel che ci serve sapere sul messaggio Telegram.
Si notino nel JSON di esempio i sottocampi del campo 'photo'; Telegram, che evidentemente è stato studiato molto bene per gestire vari formati di immagine ed evitare trasferimenti pesanti via Internet quando non è necessario, è in grado di fornire differenti risoluzioni della foto inviata, permettendoci quindi di scegliere quella "giusta", cioè quella più vicina alla risoluzione del nostro schermo, senza peraltro dover scaricare quantità enormi di dati per poi comprimerli successivamente.
Quale formato scegliere e come avviene la scelta, verrà chiarito successivamente, quando spiegheremo il decoder JPEG.
Ora prestate attenzione a un altro particolare: i campi 'file_id' contengono un codice univoco che ci permette di identificare il file da scaricare, non il nome del file stesso, che va richiesto tramite un'apposita chiamata "getFile" alle API di Telegram. La nostra libreria si occupa automaticamente di questo passaggio aggiuntivo!
Vedremo più avanti i vari campi in dettaglio; al momento ci interessa sapere come leggerli tramite Fishino, cosa non banalissima.
Su un normale PC la soluzione ovvia, e quella utilizzata da tutti, è di caricare l'intero JSON in memoria, utilizzare una delle tante librerie di "parsing" disponibili e poi leggere i campi in base al nome.
Tutto bello, tutto semplice, ma... noi abbiamo pochissima RAM a disposizione e vogliamo sfruttarla al meglio!
Il JSON dell'esempio, che è un JSON piccolo, occupa già 657 byte, corrispondenti 2/3 della memoria di una board Fishino o Arduino UNO, un decimo di quella di un Arduino/Fishino MEGA. Non poco, anzi, troppo.
Che fare quindi? L'ideale sarebbe leggere il file carattere per carattere man mano che arriva, esaminarlo al volo ed estrarne le informazioni necessarie senza memorizzare quelle inutili.
Per questo abbiamo creato la libreria **JSONStreamingParser** che realizza, come si intuisce dal nome, un parser (analizzatore, più o meno, il termine è intraducibile!) streaming, ovvero di "flusso": viene "nutrito" carattere per carattere con i dati che arrivano dal client TCP e, quando ne ha a sufficienza per capire di cosa si tratta, esegue una certa azione.
Per capire la differenza sostanziale del procedimento, vediamo uno pseudocodice che utilizza un parser normale:

Listato 4

```
void stampaValori(void)
{
 char nome[100];
 char valore[100];
 char *nomeP = nome;
 char *valoreP = valore;

 enum Stati { leggiNome, leggiValore, fine };
 Stati stato = leggiNome;

 while(stato != fine)
 {
    int c = client.read();
    switch(stato)
    {
       case leggiNome:
        // fine stringa ? (probabile errore)
        if(c  <= 0)
          // termina
          stato = fine;
        // fine nome ?
        else if(c == ':')
        {
          // si, termina il nome
          *nomeP = 0;

          // passa alla lettura del valore
          stato = leggiValore;
        }
        // carattere normale ?
        else
          // aggiunge il carattere corrente al nome
          *nomeP++ = c;
        break;
       case leggiValore:
        // fine valore o fine stringa ?
        if(c  <= 0 || c == ',')
        {
          // termina il valore corrente
          *valoreP = 0;

          // stampa nome e valore
          Serial << "NOME :" << nome << "\n";
          Serial << "VALORE:" << valore << "\n";
          // se fine stringa termina
          if( c <= 0)
            stato = fine;
          // altrimenti passa al prossimo nome
          else
          {
            // ri-inizializza i puntatori ad inizio nome e valore
            nomeP = nome;
            valoreP = valore;

            // passa allo stato di lettura nome
            stato = leggiNome;
        }
        // carattere normale ?
        else
          // aggiunge il carattere corrente al valore
          *valoreP++ = c;
        break;
       case fine:
        // stampa un messaggio di saluto
        Serial << "Ho finito di leggere tutti i valori\n";
    }
 }
}
```

- Leggo tutto il JSON dal client in una stringa di testo, chiamiamola 'json'
- Uso una classe di parsing, nella quale infilo la stringa 'json'
- Esamino i valori in base al nome

Semplificando, in C++ questo risulterebbe in qualcosa di questo tipo:

```
String json;
while(client.available())
  json += client.read();
JSONParser parser(json);
Serial << "Il nome del mittente è:" << parser.Get("first_name") << "\n";
```

Utilizzando il nostro parser, le cose cambiano radicalmente; innanzitutto dobbiamo scrivere una funzione che gestisca gli eventi JSON, come di seguito:

```
void JSONCallback(uint8_t filter, uint8_t level, const char *name, const char *value, void *cbObj)
{
   Serial << "Ho ricevuto il dato di nome " << name << " col valore " << value << "\n";
}
```

Poi occorre creare il nostro oggetto parser, collegarlo alla funzione definita sopra e "nutrirlo" con i caratteri ricevuti dal client TCP:

```
JSONStreamingParser parser;
parser.setCallback(JSONCallback, NULL);
while(client.available())
  parser.feed(client.read());
```

Il parser chiamerà AUTOMATICAMENTE la funzione JSONCallback() ogni volta che avrà ricevuto completamente un dato del tipo 'nome:valore', separando nome e valore e fornendo alcune altre informazioni, delle quali è importante la variabile 'level' che indica il livello di nidificazione del valore letto, ovvero l'indentazione (spostamento verso destra) visibile nel codice JSON ben formattato visto sopra. Potremo ricevere per esempio (sempre con riferimento al JSON riportato sopra):

```
level = 4
name = "width"
value = 51
```

Il livello risulta utilissimo per capire "dove siamo" dentro al JSON. La differenza sostanziale è

Listato 5

```
void loop(void)
{
 // process FishGram data
 FishGram.loop();

 if(millis() > tim)
 {
    DEBUG_PRINT("Free RAM : %u\n", Fishino.freeRam());
    tim = millis() + 3000;
 }
}
```

Listato 6

```
// add an event function that will be called on each received message
FishGramClass &messageEvent(FishGramMessageEvent e);

// add an event function that will be called on each received picture
FishGramClass &pictureEvent(FishGramPictureEvent e);
FishGramClass &pictureEvent(FishGramPictureEvent e, uint16_t requestedWidth, uint16_t requestedHeight);

// add an event function that will be called on each received audio message
FishGramClass &audioEvent(FishGramAudioEvent e);
```

che non saremo noi a chiedere al parser i dati, ma sarà il parser ad inviarceli, tramite la funzione JSONCallback vista sopra, quando li ha a disposizione.
Questo implica che nell'analisi non potremo richiedere un dato già passato, che è andato irrimediabilmente perso se non l'abbiamo memorizzato da qualche parte; occorre quindi studiare bene la callback in modo da poter analizzare il JSON memorizzando il minor numero di dati possibile.
Questo (ed altro) viene fatto automaticamente dalla libreria **FishGram**, tramite una cosiddetta "macchina a stati".

LE MACCHINE A STATI

Le "macchine a stati" sono un po' delle bestie nere per molti programmatori, anche se risultano molto comode in parecchie occasioni. Una spiegazione completa esula dallo scopo di questo articolo, ma ne possiamo dare una breve infarinatura con un esempio semplice-semplice.
In poche, semplici e misteriose parole, una macchina a stati è una scatola nera che riceve dati in ingresso, fornisce risposte in uscita (e questo potrebbe valere per ogni programma) ma, e questo a differenza di una funzione "normale", le risposte che dà dipendono da una o più variabili di stato interne alla funzione stessa. Inoltre, oltre a fornire una risposta in uscita, la macchina può cambiare contemporaneamente il suo stato interno.
Visto che probabilmente, come successo a chi scrive in passato, pochi avranno capito bene il discorso, facciamo un esempio molto semplice, anche se poco significativo. Diciamo di voler fare un parser semplicissimo, che riconosce una sequenza di dati di questo tipo:

nome:valore,nome:valore,.....nome:valore

da una "stringona" di caratteri. Una versione molto semplificata del parser JSON che utilizziamo.
La prima soluzione che viene in mente è di leggere tutta la stringa, poi analizzarla, cercando i ':' e le virgole, come nel programmino del **Listato 3**.

Listato 7

```
// fishgram picture event handler -- show image on TFT display and send back a confirmation message
bool FishGramPictureHandler(
 uint32_t id, const char *firstName, const char *lastName,
 FishGramRangeStream &stream, uint16_t w, uint16_t h, const char *caption)
{
 WebStream webStream(stream);
 FishinoJPEGDecoder decoder;
 decoder.setConsumer(consumeLine, 240, 320, JDR_COLOR565);
 decoder.setAutoCenter(JDR_CENTER_BOTH);
 if(decoder.setProvider(webStream) != JDR_OK)
 {
    Serial << F("Stream error\n");
    return false;
 }
 decoder
    .setAutoScale(true)
    .setRotation(JDR_ROT_AUTOCW)
 ;
 decoder.decode();

 String ans = "Ciao ";
 ans += firstName;
 ans += ", ho ricevuto la tua foto";
 FishGram.sendMessage(id, ans.c_str());

 return true;
}
```

Chiaramente abbiamo omesso qualsiasi controllo di errori, salto di spazi, eccetera. Il codice precedente fa un pesante uso dei puntatori, e risulta quindi già ben ottimizzato. Quello che si nota subito è la necessità di leggere e memorizzare l'intera "stringona" per poi analizzarla in seguito; questo implica che occorre fare una stima della sua lunghezza massima e prevedere qualche controllo sulla medesima, qui omesso per brevità.

Vediamo ora, nel **Listato 4**, la stessa cosa realizzata con una macchina a stati. Ecco qua: abbiamo messo a punto la nostra prima macchina a stati!

Il funzionamento è piuttosto semplice: la macchina inizia con lo stato 'leggiNome', si prepara quindi a leggere il nome, cosa che ci si aspetta stia ad inizio stringa.

Man mano che gli forniamo i caratteri, che come si vede vengono letti ed elaborati UNO AD UNO, e non memorizzati, la macchina modifica il suo comportamento in base a quanto ricevuto.

Mentre si trova nello stato 'leggiNome', per esempio, controlla che ci siano ancora caratteri (un valore di C nullo o negativo implica fine dati), nel qual caso passa allo stato 'fine'; se la stringa non è terminata, controlla se il carattere in arrivo è un ':', nel qual caso passa allo stato 'leggiValore'.

Infine, se non si verifica alcuno dei due casi precedenti, considera il carattere appena letto come appartenente al nome e lo aggiunge a questo.

Una volta terminata la lettura del nome, al primo ':' trovato, lo stato cambia in 'leggiValore' ed i caratteri successivi in arrivo seguiranno un'altra strada (secondo punto dello switch, case leggiValore); qui si controlla sempre se la stringa è finita (c <= 0), nel qual caso verrà comunque stampato l'ultimo valore letto, oppure che il valore sia terminato da una virgola (== ','), nel qual caso viene stampato il valore appena letto e si passa al prossimo nome oppure, se nessuno dei casi precedenti si verifica, si considera il carattere come appartenente al valore corrente al quale viene aggiunto. Il tutto termina quando viene raggiunto lo stato 'fine', nel qual caso il ciclo while() termina.

Come si può facilmente intuire, il vero vantaggio di questo modo di procedere è che si evita la memorizzazione della corposa stringa iniziale, ovvero del nostro (semplificato) JSON; si 'nutre' il tutto carattere per carattere e, quando la funzione "si accorge" di avere dati a sufficienza, stampa i risultati man mano.

Un altro vantaggio, non trascurabile, è che inserendo tutta la parte contenuta nello 'switch' in una funzione a se stante è possibile chiamarla passandole un carattere alla volta e, tra una chiamata e l'altra, far fare al nostro controller qualcos'altro! Ed infatti è proprio questo che facciamo nella funzione **loop()** del nostro sketch (**Listato 5**).

La chiamata **FishGram.loop()** esegue proprio UN SINGOLO PASSO della macchina a stati che c'è nella libreria **FishGram**; tra un passo e l'altro possiamo fare qualsiasi cosa, basta che questo 'qualsiasi cosa' duri poco tempo e non fermi il loop.

In questo caso, per esempio, stampiamo ogni 3 secondi (notate che abbiamo evitato l'uso della delay() per non fermare il loop) la memoria disponibile.

Bene... abbiamo visto che la FishGram si occupa di leggere i caratteri dal client TCP, quindi quanto viene inviato da Telegram, li passa alla **JSONStreamingParser** la quale, ad ogni 'nome:valore' che trova chiama una nostra funzione di callback; quest'ul-

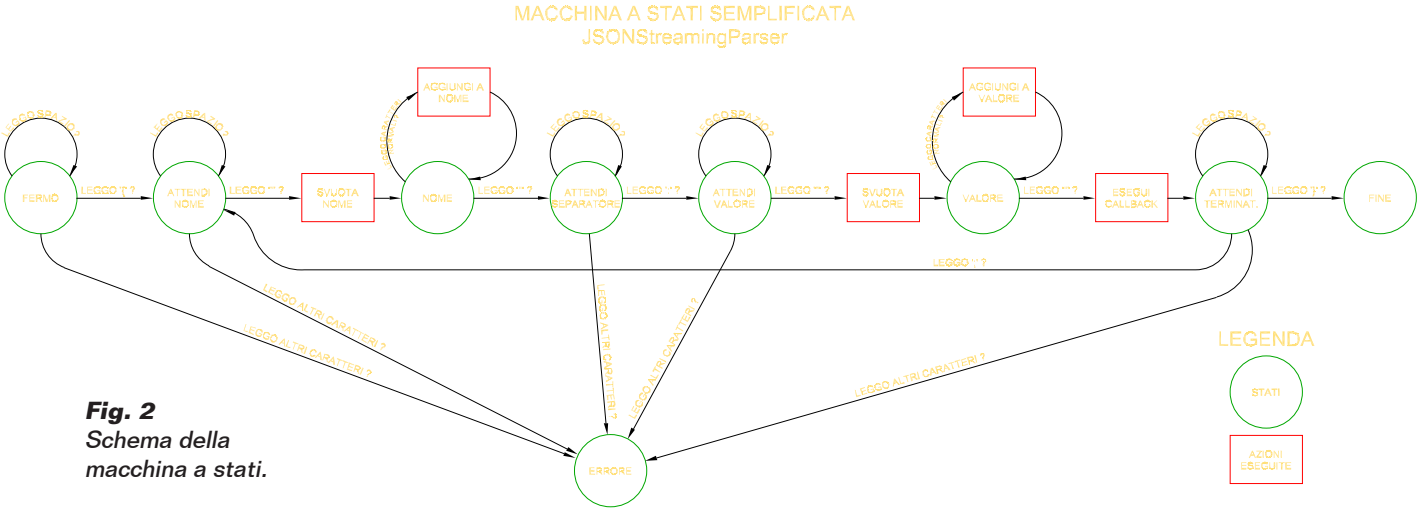


Fig. 2
Schema della macchina a stati.

tima, contenuta sempre nella libreria **FishGram**, si occupa di analizzare quanto ricevuto ed agire di conseguenza.
Nel nostro caso, la libreria **FishGram** deve:

- leggere e identificare il nome del mittente dei messaggi;
- determinare il tipo di messaggio;
- in base al tipo di messaggio, leggere e memorizzare altri valori e chiamare una funzione che si occuperà di gestire il messaggio stesso.

In dettaglio, i tipi di messaggi che ci interessano sono:

- messaggi di testo;
- immagini;
- file audio.

Abbiamo quindi previsto tre funzioni differenti per gestirli tutti distintamente; anzi, la **FishGram** permette di impostare tre funzioni esterne, ognuna destinata a gestire un certo tipo di messaggio (**Listato 6**). Queste funzioni si trovano nel file principale dello sketch, il file 'PhotoFish.ino'; vengono collegate alla **FishGram** tramite le seguenti linee della **setup()**:

```
FishGram.messageEvent(FishGramTextHandler);
FishGram.pictureEvent(FishGramPictureHandler, 240, 320);
FishGram.audioEvent(FishGramAudioHandler);
```

Vediamo in dettaglio la funzione FishGramPictureHandler(), che si occupa di gestire i messaggi contenenti immagini; il codice è quello nel **Listato 7**.
La funzione riceve i seguenti parametri:

- id = è l'id del mittente, necessario per poter rispondere;
- firstName = è il nome del mittente;
- lastName = è il cognome del mittente;
- stream = è un'oggetto che permette la lettura dei byte che compongono l'immagine (vedremo poi di cosa si tratta);
- w = la larghezza dell'immagine;
- h = l'altezza dell'immagine;
- caption = un eventuale titolo/commento assegnato all'immagine.

In breve, si occupa di leggere i dati dell'immagine provenienti dal client web, decodificarne il formato jpeg, inviarli al display ed infine rispondere al messaggio ricevuto con una conferma di ricezione. Non male per poche linee di codice!
Vediamo in dettaglio, sempre con un occhio sullo schema a blocchi, tutti gli elementi utilizzati nella funzione.

FishGramRangeStream

Uno dei (numerosi) ostacoli riscontrati nello sviluppo dell'applicazione, anzi, il più grosso, è stato quando ci siamo resi conto che il client HTTPS del modulo ESP ha un buffer di ricezione limitato a 8 kB massimi; questo è stato voluto per contenere l'uso della preziosa memoria RAM nel medesimo.
Ebbene, il server di Telegram, seguendo uno standard più o meno consolidato, considera che chi riceve utilizzi un buffer di 16 kbyte, quindi il doppio, ed invia di conseguenza pacchetti HTTPS codificati in blocchi di quella dimensione, quando deve trasmettere grosse moli di dati.
Quindi, se richiediamo un file di dimensioni inferiori a circa 7 kbyte tutto va bene, mentre per dimensioni superiori il modulo WiFi va in errore e chiude la connessione TCP senza ricevere nulla.
Questo scoglio stava per farci rinunciare all'applicazione quando, dopo una sana ricerca in rete, abbiamo "scoperto" che molti server web sono capaci di "spezzettare" a richiesta gli invii di file; per la precisione, sono in grado di inviare parti di file delimitate da un inizio ed una fine.
Questo non per risolvere il nostro problema, ma per poter permettere ai browser web di riprendere uno download interrotto per motivi di connessione!
Il trucchetto si ottiene aggiungendo un header http di questo tipo alla richiesta:

Range: byte=inizio-fine

dove 'inizio' e 'fine' sono due numeri che indicano la parte di file che ci interessa.
Fortunatamente il server di Telegram supporta questa estensione, quindi abbiamo scritto una classe apposita, appunto la **FishGramRangeStream**, che si occupa di "spezzettare" le richieste più grandi di 4 kByte in "monconi" di quella dimensione, in maniera trasparente.
Vediamo il relativo costruttore:

```
FishGramRangeStream::FishGramRangeStream(FishinoSecureClient &client,
const char *token, bool flashToken, const char *path, uint32_t size);
```

Per creare un'oggetto di questa classe occorre passare il client, la token del nostro bot Telegram, il percorso del file remoto da scaricare (relativo al nostro bot) e la dimensione del file stesso, fornito da Telegram insieme ai dati del messaggio.
Una volta creato l'oggetto per leggere i dati basta richiamare la funzione **read()**, che si comporta esattamente come la **read()** del client; lo spezzettamento delle richieste avviene in modo completamente trasparente, così come l'apertura e la chiusura del

client e la richiesta del file stesso.
L'oggetto **FishGramRangeStream** viene comunque creato dalla classe **FishGram** e passato alla nostra funzione; a noi non resta altro che utilizzarlo per leggere il file remoto.

WebStream

Un altro "stream" direte! A che serve ? Non bastava il **FishGramRangeStream**?
Sviluppando una libreria software cerchiamo sempre di curare due cose:

- la modularità;
- l'indipendenza da altre librerie, dove possibile.

Come vedremo in seguito, nell'analisi dettagliata del decoder JPEG, anche quest'ultimo lavora leggendo un "flusso" (stream) di dati, decodificandoli al volo ed inviandoli al display (o ad altro) senza memorizzare nulla.
Quindi ha la necessità di un oggetto di tipo stream, "agganciabile" ai vari supporti che possono spaziare dalla memoria SD ad un client Web.
La libreria **FishinoJPEGDecoder** contiene quindi la definizione di una classe generica **FishinoJPEGDecoderStream** utilizzata, appunto, per leggere i dati da decodificare. Nella libreria è già prevista una classe derivata da quella in grado di leggere da un oggetto di tipo File, quindi dalla scheda SD, ed un'altra classe derivata per leggere le immagini memorizzate direttamente nella memoria Flash.
Però a noi occorre un oggetto che faccia da "ponte" tra un **FishinoJPEGDecoderStream** ed un **FishGramRangeStream**; abbiamo quindi creato, direttamente nello sketch, una piccola classe WebStream in grado di assolvere tale compito. Questa classe, derivata dalla **FishinoJPEGDecoderStream**, è in grado di leggere dati provenienti dalla **FishGramRangeStream**.
Riepilogando:

- il decoder JPEG (**FishinoJPEGDecoder**) richiama la **WebStream** per leggere i dati;
- la WebStream richiama la **FishGramRangeStream** per leggere i dati;
- la **FishGramRangeStream** si occupa di spezzettare le richieste ed inviarle a Telegram tramite il client TCP.

Il tutto adesso dovrebbe essere (quasi) chiaro rotonando indietro di qualche pagina e osservando lo schema a blocchi della nostra applicazione, proposto nella **Fig. 1**.

Listato 8

```
#include <FishinoGFX.h>
#include <FishinoILI9341.h>
#include<FishinoJPEGDecoder.h>
#include <SD.h>

#ifdef SDCS
 #define SD_CS SDCS
#else
 #define SD_CS 4
#endif

// funzione di callback
bool consume(JDR_RECT const &rec, uint8_t const *buf)
{
 t.setAddrWindow(rec.left, rec.top, rec.left + rec.width - 1, rec.top + rec.height - 1);
 uint16_t const *buf16 = (uint16_t const *)buf;
 FishinoILI9341.pushColors(rec.width * rec.height, buf16);
 return true;
}

// esempio di decoding JPEG da file su scheda SD
void setup()
{
 FishinoILI9341.begin();
 FishinoILI9341.fillScreen(ILI9341_BLUE);

 SD.begin(SD_CS);

 FishinoJPEGDecoderFileStream s;
 if(!s.open("chri8s.jpg"))
    DEBUG_PRINT("Error opening file\n");
 else
 {
    FishinoJPEGDecoder decoder;
    decoder.setConsumer(consume, 240, 320, JDR_COLOR565);
    decoder.
       setAutoCenter(JDR_CENTER_BOTH)
       .setAutoScale(true)
       .setRotation(JDR_ROT_AUTOCW)
    ;
    decoder.setProvider(s);
    decoder.decode();
 }
}
```

IL DECODER JPEG, OVVERO LA LIBRERIA FishinoJPEGDecoder

Lo sviluppo del decoder JPEG è stato la parte più complessa dell'applicazione, anche perché si è cercato di rendere la libreria il più possibile generica e funzionante con la quasi totalità dei JPEG disponibili. Diciamo subito che "non è tutta farina del nostro sacco", perché per lo sviluppo siamo partiti da una piccola ma interessante libreria open source reperibile in Internet al seguente indirizzo:

http://elm-chan.org/fsw/tjpgd/00index.html

La libreria in oggetto è resa disponibile con una licenza estremamente aperta (stile BSD, con in aggiunta la possibilità di evitare di indicare la licenza quando si distribuisce il solo codice binario); è quindi l'ideale per il nostro utilizzo, non ponendo praticamente vincoli legali.

La libreria, pur essendo ben studiata e soprattutto ben commentata, ha qualche limite che abbiamo superato con il nostro codice:

- non gestisce un particolare formato di oversampling delle componenti cromatiche (ce ne siamo accorti tentando di decodificare alcuni file provenienti da una macchina digitale e successivamente ruotati);
- non è in grado di ruotare immagini decodificate;
- non è in grado di scalare immagini decodificate;
- è scritta in linguaggio C puro, scomodo da utilizzare in ambiente Arduino, che, lo sappiamo, si programma utilizzando un C personalizzato e semplificato.

L'aggiunta del formato mancante è stata la parte più semplice, come vedremo in seguito.
Rotazione e scaling delle immagini, per contro, hanno comportato la riscrittura di una buona fetta di codice originale, del quale abbiamo mantenuto in pratica la parte di lettura degli header JPEG, la decompressione Huffman e la DCT (Discrete Cosine Transformation, trasformata coseni inversa) che fanno parte dell'algoritmo JPEG.
Già che c'eravamo, scrivendo la parte di scaling/rotazione, abbiamo aggiunto un automatismo che permette di adattare l'immagine ad una dimensione di display specificata, calcolando rotazione e fattore di scala in modo automatico.
Il tutto è stato, infine, incapsulato in una classe C++ facilmente utilizzabile in ambiente Arduino.
Vediamo per prima cosa l'utilizzo della libreria, lasciando la descrizione teorica del formato JPEG e del codice utilizzato alla seconda e ultima puntata dell'articolo.

USO DEL DECODER

Iniziamo subito con uno spezzone di codice (**Listato 8**) in grado di decodificare e visualizzare un'immagine contenuta in un file su scheda SD, spiegandone i vari punti.
Come abbiamo accennato in precedenza, anche il nostro decoder lavora in modalità "streaming", per poter funzionare con una quantità minima di memoria RAM. Questo significa che la libreria legge man mano il file jpeg, lo decodifica "al volo" e spedisce quanto decodificato al display, in blocchi rettangolari di una certa dimensione.
In questo modo l'unica cosa da memorizzare dell'immagine è il blocco rettangolare corrente, che al massimo è di 16x16 pixel, ma può anche essere di un singolo pixel.
Vediamo in dettaglio il codice nella setup(), tralasciando le ovvie inizializzazioni di scheda SD e display TFT: la creazione dell'oggetto decoder è:

```
FishinoJPEGDecoder decoder;
```

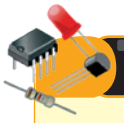
Poi si collega la funzione 'consumer':

```
decoder.setConsumer(consumeLine, 240, 320, JDR_COLOR565);
```

La funzione 'consumer' è una funzione di callback, ovvero che viene chiamata automaticamente dalla libreria quando questa ha "qualcosa di pronto", ovvero un blocco di pixel decodificato e pronto per essere visualizzato.
Nel nostro caso la funzione si chiama 'consume', ed è la seguente:

```
// funzione di callback
bool consume(JDR_RECT const &rec, uint8_t const *buf)
{
   t.setAddrWindow(rec.left, rec.top, rec.left + rec.width - 1, rec.top + rec.height - 1);
   uint16_t const *buf16 = (uint16_t const *)buf;
   FishinoILI9341.pushColors(rec.width * rec.height, buf16);
   return true;
}
```

La funzione riceve come parametri un "rettangolo", ovvero le coordinate dei pixel sul display che sono in arrivo, ed un buffer di byte contenente i dati stessi dei pixel.
Il formato di questi dati dipende da quello che abbiamo richiesto usando la funzione **setConsumer()**; nel nostro caso abbiamo richiesto il formato JDR_COLOR565, che è un formato a 16 bit (due byte), composto da 5 bit per il colore rosso, 6 bit per il colore verde ed altri 5 bit per il colore blu. Sono visualizzabili quindi 65.535 differenti



per il MATERIALE

Tutti i componenti utilizzati in questo progetto sono di facile reperibilità. Il master del circuito stampato può es della rivista così come il f programmare il microcont ensore ad infrarossi IR38D entre l'integrato Microchip MCP3905A è disponibile 4,20 Euro.

Sostituire testo

Il materiale va richiesto a:
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)
Tel: 0331-799775 • http://www.futurashop.it

colori/sfumature, che è quanto supportato dal nostro display nella modalità prescelta.
La libreria supporta altri formati, per esempio il JDR_COLOR888, che consiste in 8 bit per ogni colore, il cosiddetto formato "True Color", in grado di rappresentare ben 16.777.216 colori differenti (ossia in True Color, a 16,7 milioni di colori) ossia le immagini con una profondità di colore sufficiente a coprire tutte le sfumature percettibili dall'occhio umano.
Abbiamo aggiunto inoltre altri formati meno usati ma che potrebbero essere utili: un formato ad 8 bit per pixel (JDR_COLOR323), abbastanza crudo come qualità (solo 256 colori disponibili, senza tabella di conversione), un formato con 256 livelli di grigio, JDR_GRAYSCALE e, per ultimo, un formato in bianco/nero con un bit per pixel, nel caso si volesse usare la libreria con un display grafico monocromatico. Tornando alla funzione consumer, questa si limita ad impostare una "finestra di lavoro" sul display:

```
t.setAddrWindow(rec.left, rec.top, rec.left + rec.width - 1, rec.top + rec.height - 1);
```

e a copiarci i pixel :

```
uint16_t const *buf16 = (uint16_t const *)buf;
FishinoILI9341.pushColors(rec.width * rec.height, buf16);
```

Nella nostra **setup()**, una volta impostata la funzione 'consume' è necessario specificare qualche parametro di conversione:

```
decoder.
        setAutoCenter(JDR_CENTER_BOTH)
        .setAutoScale(true)
        .setRotation(JDR_ROT_AUTOCW)
        ;
```

Qui chiediamo al nostro decoder di centrare l'immagine sia orizzontalmente che verticalmente (JDR_CENTER_BOTH), di scalarla automaticamente (setAutoScale(true)) e di ruotarla automaticamente, se necessario, in modo da sfruttare al massimo il display, scegliendo il senso orario in caso di rotazione (JDR_ROT_AUTOCW).
Per ultimo, impostiamo il "fornitore" di dati, ovvero lo stream da cui provengono i medesimi. In questo caso, leggendo da SD, abbiamo creato uno stream del tipo FishinoJPEGDecoderFileStream, in grado, appunto, di lavorare con file. Nel nostro applicativo PhotoFish, invece, abbiamo utilizzato un oggetto stream in grado di leggere direttamente i dati da un client web. L'impostazione del provider avviene con la linea seguente:

```
decoder.setProvider(s);
```

Qui per brevità non abbiamo controllato il codice di errore ritornato dalla funzione setProvider() che, se differente da JDR_OK, indica un errore nei dati dell'immagine. Impostando il provider, infatti, la libreria inizia l'analisi dell'immagine, ne estrae le dimensioni, il tipo di compressione, eccetera.
Per ultimo, lanciamo la decodifica:

```
decoder.decode();
```

Questa semplice linea avvia la decodifica; la libreria inizierà a leggere i dati dal "provider", li elaborerà, e man mano che avrà pixel da visualizzare eseguirà le chiamate necessarie alla 'consumer', fino ad esaurimento immagine. Anche questa funzione ritorna un codice che, se è diverso da JDR_OK, indica un qualche tipo di errore.

CONCLUSIONI

Bene, terminiamo qui la prima parte della trattazione, nella quale vi abbiamo esposto l'applicazione PhotoFish e le librerie utilizzate nella scrittura dello sketch allo scopo di gestire i messaggi di Telegram e visualizzare le immagini sul display TFT. Avete così avuto modo di scoprire come, con risorse di calcolo e di memoria relativamente limitate come quelle delle board Arduino/Fishino, è possibile interagire con Telegram per ricevere e mostrare immagini su uno shield TFT.
Nella prossima puntata, conclusiva del progetto, approfondiremo il discorso sui formati di immagine e sulle compressioni video, nonché sulla codifica e la decodifica. Lì vi troverete un approfondimento sull'elaborazione delle immagini, come la rotazione, lo scaling e altri concetti accennati nelle pagine precedenti; spiegheremo bene anche come avviene la compressione JPEG, molto utilizzata nei contenuti web.
Per utilizzare il progetto, in questa prima puntata vi abbiamo fornito le nozioni di base, che potrete sfruttare da subito.