Sergio Moro (nicsergio)

DOMOTICA DIY: RASPBERRY COME INTERFACCIA UTENTE PER ARDUINO

10 December 2020

Sto frequentando realmente il forum da poco, da quando ho scoperto la sezione degli articoli: un mare di informazioni utili e spesso molto ben spiegate, un piccolo "tesoro" in cui con sempre maggiore frequenza mi perdo "piacevolmente". Ora, il mio articolo non ha ovviamente la pretesa di fornire informazioni utili, potrebbe aiutare magari chi fatica a prendere sonno o rallegrare l'umore a chi ha la pazienza di leggere le mie castronerie, volevo però portare anche il mio insignificante contributo.

1. Premessa

Al giorno d'oggi, per rendere la propria casa un po' più "smart", automatizzare alcune funzioni, governare dispositivi da remoto o contenere i consumi, ci sono una pletora di soluzioni, da quelle più costose e flessibili, a quelle hobbistiche alla portata di tutti.
Alcuni appassionati (chiamati a volte simpaticamente "smanettoni") seguono la strada più difficile: provare a fare qualcosa interamente da sé, dando appunto sfogo alla loro passione, accettando i pro ed i contro rispetto ad adottare una soluzione bella e pronta.
Dopo alcune titubanze, in questa pagina voglio raccontarvi una parte del progetto che ho portato avanti nei ritagli tempo, nell'arco di quasi due anni, completamente da autodidatta.
Questo articolo è una umile condivisione, non avendo io background in materia: prima di questa esperienza non avevo mai utilizzato un Raspberry o un Arduino o scritto delle pagine web e avevo una conoscenza pressoché nulla di linguaggi come C++, PHP e javascript, le soluzioni adottate saranno banali per molti ed esisteranno sicuramente metodi più efficienti, raffinati e puliti per fare le stesse cose. Nonostante questo il tutto funziona bene e stabilmente da quattro anni (non lo nego, con un po' di incredulità ed una certa soddisfazione) e tuttora mi diverto ad ampliare o ritoccare alcune funzionalità.

2. Uno sguardo all'hardware

Ecco lo schema di massima delle interconnessioni:

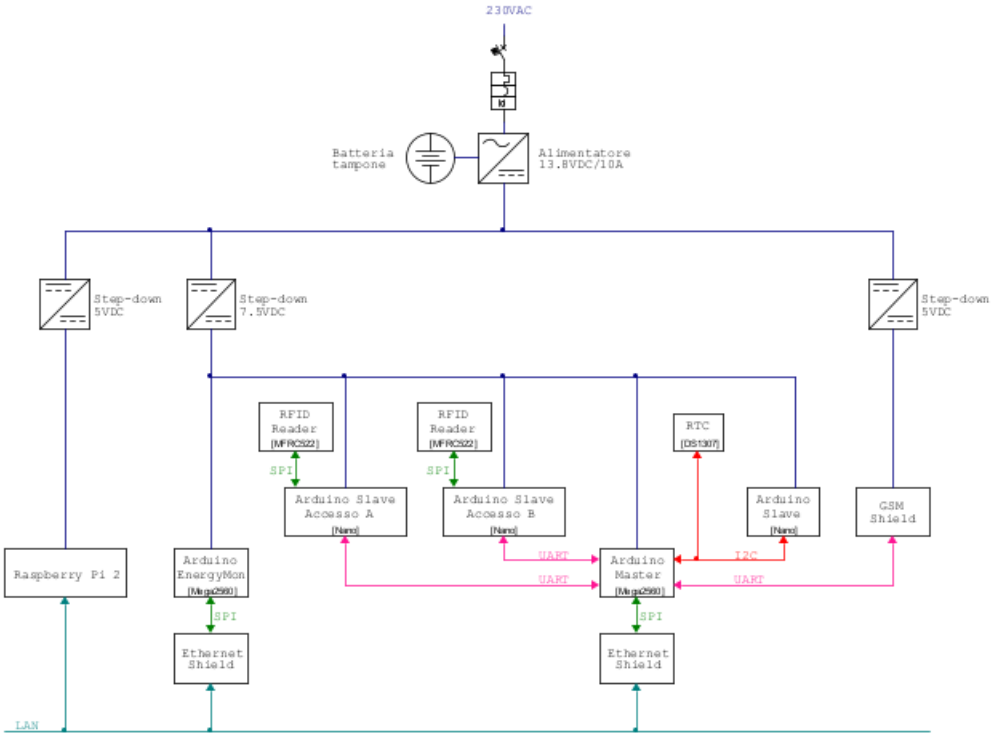

Figura 1 - Schema connessioni

Sul web gli esempi di connessione tra Raspberry e Arduino che si trovano sono tutti via seriale, nel mio caso ho preferito evitare dovendo coprire una distanza "importante" (più di 20 metri) ed avendo esaurito le porte UART sull'Arduino Master, così ho preferito una connessione Ethernet, che mi ha consentito di interrogare direttamente Arduino da un PC in LAN durante le operazioni di debug, ancora prima di avere disponibile il webserver. Aggiungo che sono disponibili schede di rete con interfaccia SPI molto compatte ed a costi irrisori, come la WIZ850io:

Figura 2 - WIZ850io

2.1 Arduino

La parte di logica del comando dispositivi e di raccolta dei segnali dal campo è svolta da degli Arduino, a cui fa capo un Arduino Master connesso in rete LAN, qui abbiamo principalmente l'implementazione di un sistema di allarme, comandi centralizzati/automatici dei frangisole, gestione dell'irrigazione, apertura cancello e comando termobagni. Tempo dopo ho aggiunto in rete un secondo Arduino per il monitoraggio dell'energia assorbita dall'abitazione e quella prodotta dai pannelli fotovoltaici (maggiori info su OpenEnergyMonitor.org).

Ho scelto la piattaforma Arduino per la velocità di apprendimento, l'IDE è però piuttosto limitato, per cui ho utilizzato fin da subito Visual Studio con il plugin Visual Micro e più recentemente sono passato a Visual Studio Code con PlatformIO.

2.2 Raspberry

Gli obiettivi per l'interfaccia utente erano:

- poter interagire col sistema, inviando comandi, leggendo stati e cambiando impostazioni
- disponibilità dell'interfaccia sui vari dispositivi in rete LAN (PC, smartphone, tablet)
- disponibilità dell'interfaccia da remoto
- disponibilità immediata dell'interfaccia su un pannellino fisso
- possibilmente non dover scrivere APP specifiche per questo o quel sistema operativo

L'idea era quella di realizzare delle pagine web, fruibili quindi da qualunque dispositivo attraverso un comune browser, certo un webserver è realizzabile anche direttamente su Arduino, ma volevo qualcosa di più flessibile e, soprattutto, un pannello a disposizione senza dover ogni volta smanettare con lo smartphone.
La scelta è caduta sul piccolo Raspberry Pi 2 (eravamo nel 2015) che veniva comodo anche per far girare, oltre al webserver per l'interfaccia, anche altri servizi ad esempio di raccolta dei dati energetici (EmonCMS).
Per assolvere le funzioni di pannellino bastava dotarlo di un display touch e sarebbe potuto divenire anche una simpatica cornice digitale, avrebbe potuto visualizzare automaticamente la telecamera esterna al suono del campanello e così via..
Ho quindi acquistato un kit cinese veramente economico (costo sui 30€), molto base, ma per questo uso sufficiente, comprendente:

- display LCD 7" AT070TN90 risoluzione 800x480
- scheda di controllo VS-TY2662 con ingressi HDMI, VGA ed RCA composito
- pannello touch resistivo TT070TP da connettere con adattatore USB

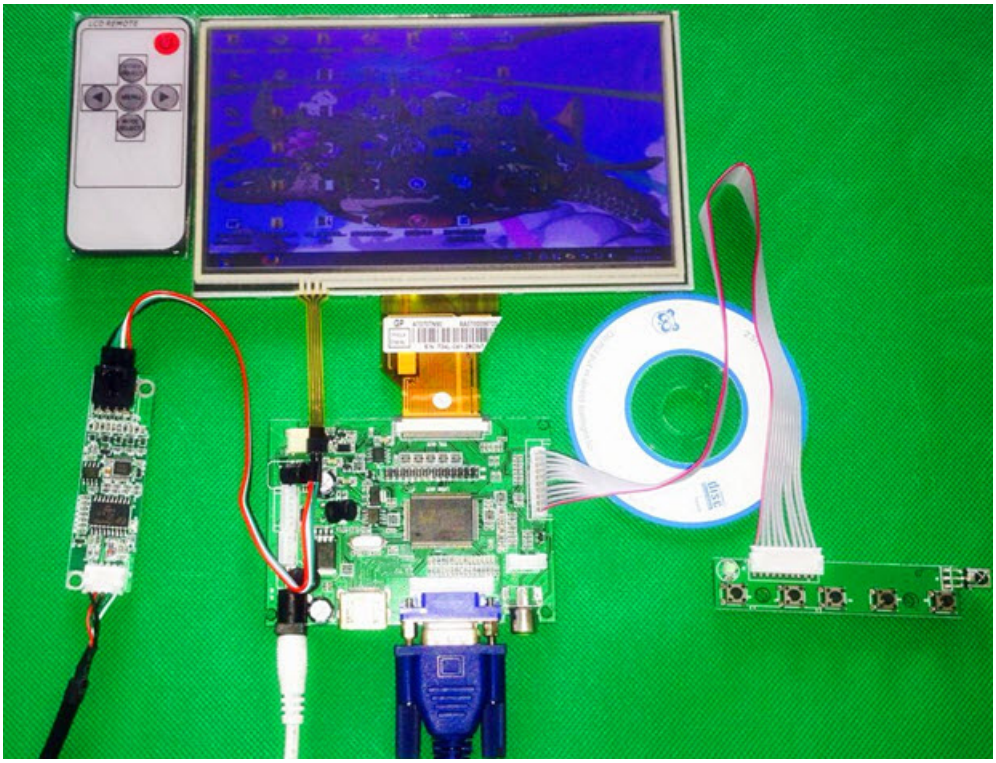

Figura 3 - Display

Con un po' di ingegno lo ho fissato ad una parete in cartongesso:

Figura 4 - Pannello

ed ho utilizzato un pulsante libero (a contatto pulito) del videocitofono a fianco per accendere/spegnere il display.

3. La comunicazione

Lato Arduino ho implementato un socket server, sempre in ascolto di eventuali richieste, mentre lato Raspberry/webserver abbiamo uno script PHP che crea un client il quale si connette al socket server, invia la richiesta e restituisce la risposta.

La sintassi del messaggio scambiato è molto semplice, i dati della richiesta e della risposta sono trasmessi tramite una stringa con terminatore nullo (la classica C-string), la quale ha una parte iniziale a lunghezza fissa, di 10 caratteri, che contiene il codice di comando <XXX> o della risposta <YYY>, seguito dagli eventuali parametri, separati da "|".

Esempio di richiesta:

```
//cmd#XXX!Par1|Par2|Par3
```

Esempio di risposta:

```
//ret#YYY!Par1|Par2|Par3
```

Il codici di comando sono parecchi, ovviamente ogni codice di comando ha la sua implementazione lato Arduino, può o meno richiedere dei parametri aggiuntivi e può o meno prevedere dei parametri di risposta.

Il codice di risposta è un enumerativo:

- 001 -> OK: comando eseguito
- 100 -> Errore: sintassi comando non valida
- 101 -> Errore: codice comando non riconosciuto
- 102 -> Errore: parametri aggiuntivi comando non corretti

Per evitare di dover ricaricare completamente le pagine ad ogni interazione, ho utilizzato il meccanismo AJAX, che consente di effettuare richieste asincrone al server con aggiornamento dinamico delle pagine, il giro potrebbe essere schematizzato così:

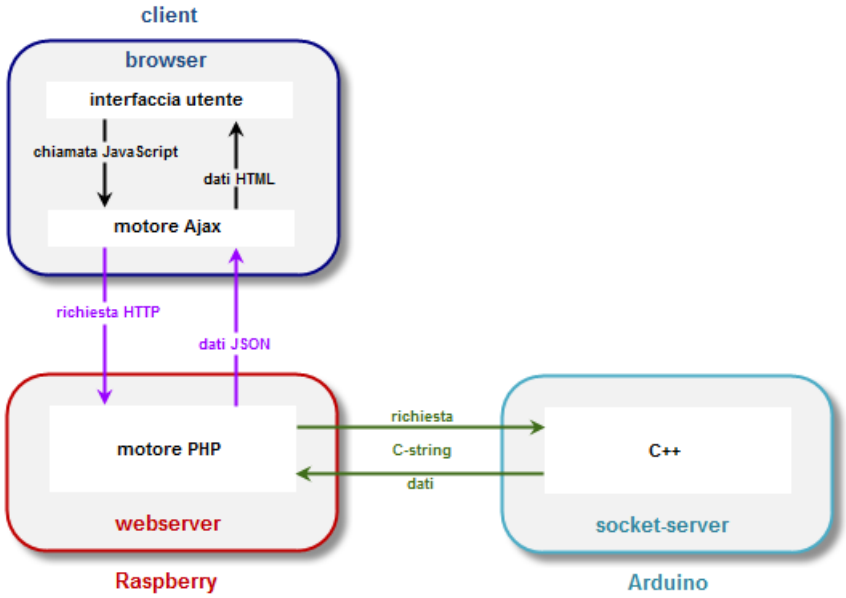

Figura 5 - Schema comunicazione

3.1 Lato Arduino

Il codice che implementa il socket server è incapsulato in una classe, tra i membri privati abbiamo l'oggetto server, la stringa di appoggio di 128 caratteri che, per risparmiare memoria, viene utilizzata sia per memorizzare il messaggio inviato dal client che la risposta da restituire e l'oggetto PString, usato per manipolare comodamente la stringa:

```
private:
   //Server Web
   EthernetServer*    _webServer;

   //Dati trasmessi via Web
   char               _webData[128];

   //Manipolatore stringa dati
   PString*           _webStr;
```

in fase di inizializzazione vengono eseguite le seguenti istruzioni:

```
//Inizializzazione stringa
memset(&_webData, 0x0, sizeof(_webData));

//Istanza manipolatore stringa dati
_webStr = new PString(_webData, sizeof(_webData));
```

```
//Istanza web server
_webServer = new EthernetServer(WebPort);

//Inizializzazione server
_webServer->begin();
```

il metodo *update()* è richiamato ciclicamente e, in seguito alla ricezione di una richiesta, popola la stringa *_webData* con il messaggio inviato dal client, richiama il metodo privato *onWebData()* che gestisce il comando richiesto ed infine restituisce la riposta al client:

```
//Aggiornamento istanza interfaccia
void WebInterface::update()
{
   EthernetClient webClient = _webServer->available();

   //Se un client è connesso
   if (webClient)
   {
      //Se il client invia dati
      if (webClient.connected() && webClient.available())
      {
         //Puntatore ad inizio stringa
         _webStr->begin();
         char c;
         //Iterazione di lettura dati inviati dal client (atteso terminatore)
         while (webClient.available())
         {
            //Lettura byte
            c = webClient.read();
            if (c != '\0')
               //Accodamento carattere
               _webStr->print(c);
            else
               //Fine memorizzazione
               break;
         }
         //Eventuale svuotamento buffer
         while (webClient.available())
            webClient.read();

         //Processazione evento di ricezione dati
         onWebData();
         //Invio della risposta al client (stringa con terminatore)
```

```
            webClient.write((const uint8_t *)&_webData, strlen(_webData) + 1);

            webClient.stop();
        }
    }
}
```

il metodo *onWebData()* effettua un primo controllo sulla sintassi del messaggio ed invoca la funzione di callback per l'esecuzione del comando:

```
void WebInterface::onWebData()          //Evento di ricezione dati
{
    //Verifica sintassi per richiesta comando via Web
    if (!strncmp_P(_webData, PSTR("//cmd#"), 6) &&
         _webData[HeaderLen - 1] == '!')
    {
        //Inizializzazione per accesso a primo parametro di comando
        _fstCmdPar = true;
        //Inizializzazione per accesso a primo parametro di risposta
        _fstRetPar = true;
        //Risposta non inizializzata
        _replyInitialized = false;

        char cmdCode[4];
        //Valorizzazione codice comando richiesto (e terminazione)
        strncpy(cmdCode, &_webData[6], 3); cmdCode[3] = '\0';

        //Esecuzione comando richiesto (e composizione testo di risposta)
        webRet_t ret = (*_webCmdExeFct)((webCmd_t)atoi(cmdCode));

        //Se comando non eseguito o risposta non inizializzata
        if (ret != RetOK || !_replyInitialized)
            initReply(ret);
    }
    else
        //Composizione testo di risposta: sintassi non valida
        initReply(RetErrSyntax);
}
```

nella libreria sono definiti i seguenti enumerativi:

```
enum webCmd_t{                          //Tipo di dato: codice comando
              WebTimeRead    = 001,     //--> <  lettura data/ora
              WebTimeSet     = 002,     //-->  > impostazione data/ora
```

```
                WebAutTimeRead = 005,  //--> <  lettura parametri per gestione aut. orolo
                WebAutTimeSet  = 006,  //--> <> impostazione parametri per gestione aut.
                ...
                ...
};

enum webRet_t{                         //Tipo di dato: codice risposta
                RetOK          = 001,  //--> comando eseguito
                RetErrSyntax   = 100,  //--> sintassi comando non valida
                RetErrCmdCode  = 101,  //--> codice comando non riconosciuto
                RetErrPars     = 102   //--> parametri aggiuntivi comando non corretti
};
```

ed il puntatore a funzione *webCmdExe* da predisporre per l'implementazione dei comandi:

```
//Puntatore a funzione per l'esecuzione del comando, la funzione
//ha come ingresso il codice di comando richiesto e restituisce
//il codice di risposta da inviare al client

typedef webRet_t (*webCmdExe)(webCmd_t);
```

il costruttore della classe prevede di passare il puntatore alla funzione predisposta per l'esecuzione dei comandi, che viene assegnato al membro della classe in fase di inizializzazione.

Il comando viene infine gestito nella funzione di callback predisposta nel programma, ad esempio:

```
//Analisi ed esecuzione comando remoto
webRet_t webCmdExecute(webCmd_t webCmd)
{
   char* par = NULL;

   //Selezione comando richiesto
   switch (webCmd)
   {
      //Lettura data/ora
      case WebTimeRead:
         //Valorizzazione parametri di risposta
         webInt.addRetPar(rtc.hour());
         webInt.addRetPar(rtc.minute());
         webInt.addRetPar(rtc.second());
         webInt.addRetPar(rtc.day());
         webInt.addRetPar(rtc.month());
         webInt.addRetPar(rtc.year());
         //Comando eseguito
```

```
            return RetOK;

        //Impostazione data/ora
        case WebTimeSet:
            {
                tmel_t time;
                //Iterazione compilazione struttura TimeElements
                for (byte i = 0; i < 6; i++)
                {
                    //Lettura parametro di comando
                    par = webInt.getCmdPar();
                    if (!par)
                        return RetErrPars;
                    //Assegnazione TimeElements
                    switch (i)
                    {
                        case 0: time.Hour   = atoi(par); break;
                        case 1: time.Minute = atoi(par); break;
                        case 2: time.Second = atoi(par); break;
                        case 3: time.Day    = atoi(par); break;
                        case 4: time.Month  = atoi(par); break;
                        case 5: time.Year   = atoi(par); break;
                    }
                }
                //Impostazione data/ora
                rtc.adjust(&time);
            }
            //Comando eseguito
            return RetOK;

        case ...:

        case ...:

    }
}
```

mentre la classe rende disponibile il metodo pubblico *char* getCmdPar()* per accedere ai parametri aggiuntivi relativi al comando e *addRetPar(x)* per valorizzare eventuali parametri di risposta.

3.2 Lato Raspberry

3.2.1 Javascript

Per le pagine web ho utilizzato la libreria jQuery, per comodità e perché serviva anche per la tastiera virtuale, il codice Javascript che si occupa di inviare una richiesta al server è il seguente:

```
function socket(successFct, server, command, parameters)
{
   // Comunicazione con Arduino, parametri:
   //  successFct: puntatore a funzione da eseguire in caso di successo
   //  server:     indice selezione server
   //  command:    codice comando
   //  parameters: eventuale array parametri aggiuntivi

   if (server === undefined)
      server = 0;                  //--> selezione server principale

   $.ajax(                         //Richiesta asincrona al server
   {
      type:     "GET",
      url:      "php/socket.php", //--> script lato server
      data:     {srv: server, cmd: command, pars: parameters},
      dataType: "json",           //--> tipo dati restituiti
      timeout:  "7500",
      success:  function(data)    //Callback se richiesta eseguita
      {
         if (data.error)          //Se restituito un errore
             notify("Errore " + data.source +
                    " (" + data.code + "): " + data.message);
         else
         {
            notify();             //--> pulizia notifiche
            if (successFct)
               successFct(data);  //--> esecuzione callback
         }
      },
      error:   function(jqXHR, exception) //Callback con errori
      {
         //Errore con evento di cambia pagina e AJAX in corso
         if (jqXHR.status === 0)
            return;               //--> nessuna segnalazione
         //Notifica errore richiesta asincrona
```

```
            notify("Errore AJAX: " + exception);
        }
    });
}
```

il metodo *notify(error)* non fa altro che visualizzare nella barra a fondo pagina il testo dell'errore in colore rosso.
Sono poi disponibili, sempre nello stesso file .js, i metodi *socketDly* e *socketLoop* per effettuare la richiesta con un certo ritardo e per iterarla ad intervalli di tempo (refresh automatico).

3.2.2 PHP

Lato webserver, viene invocato lo script *socket.php*:

```
<?php
/*-------------------------------------------------------------
Comunicazione con Arduino (socket server) da richiesta HTTP GET
Parametri in ingresso:
       srv:   selezione server
              0 --> Arduino principale
              1 --> Arduino monitoraggio energetico
       cmd:   codice comando
       pars:  parametri relativi al comando
Parametri in uscita:
       stringa JSON dei dati di ritorno dal server
-------------------------------------------------------------*/

//Inclusione libreria di comunicazione con socket server Arduino
include 'socket.lib.php';

//Esecuzione comando & restituzione stringa JSON
echo json_encode(socketRequest($_GET['srv'], $_GET['cmd'],
                 isset($_GET['pars']) ? $_GET['pars'] : NULL));
?>
```

il file *socket.lib.php* contiene il codice vero e proprio che si occupa della comunicazione con Arduino:

```
function socketRequest($server, $command, $params = NULL)
{
   /*-----------------------------------------------------------
   Gestione richiesta socket al server
   Parametri in ingresso:
          server:   selezione server
```

```
        command:  codice comando
        params:   parametri relativi al comando
Parametri in uscita:
        array dati richiesti
------------------------------------------------------------*/

$sendTimeout = array('sec'=>1, 'usec'=>500000);
$recvTimeout = array('sec'=>1, 'usec'=>500000);
$retSyntax   = '//ret#XXX!';
$sep         = '|';

error_reporting(E_ALL);

//Creazione socket
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP)
                        or die(socketError());
//Impostazione timeout invio
socket_set_option($socket, SOL_SOCKET, SO_SNDTIMEO, $sendTimeout);
//Impostazione timeout ricezione
socket_set_option($socket, SOL_SOCKET, SO_RCVTIMEO, $recvTimeout);
//Connessione
socketConnect($socket, $server) or die(socketError());

//Creazione stringa messaggio con sintassi comando
$message = sprintf('//cmd#%03d!', $command);
if (isset($params))
{
   if (is_array($params))
      $message .= implode($sep, $params);
   else
      $message .= $params;
}
$message .= "\0";    //--> terminatore

//Invio dati (Arduino attende una C-string)
socket_write($socket, $message, strlen($message))
             or die(socketError());
//Ricezione dati (Arduino invia una C-string)
$result = socket_read($socket, 128, PHP_BINARY_READ)
                       or die(socketError());
$result = rtrim($result, "\0");
socket_close($socket);
```

```
   //Verifica sintassi risposta
   $retLen = strlen($retSyntax);
   if (!strncmp($result, $retSyntax, $retLen - 4) &&
       $result[$retLen - 1] === $retSyntax[$retLen - 1])
   {
      //Codice risposta da Arduino
      $retCode = intval(substr($result, $retLen - 4, 3));

      //Se comando eseguito
      if ($retCode === 1)
      {
         //Se presenti parametri di risposta
         if (strlen($result) > $retLen)
            //Restituzione dei dati come array
            return explode($sep, substr($result, $retLen));
      }
      else
         //Segnalato errore da Arduino
         die(arduinoError($retCode));
   }
   else
      //Sintassi risposta non valida
      die(arduinoError(200));
}
```

la funzione di connessione al socket-server:

```
socketConnect($socket, $server)
{
   /*-------------------------------------------------------------
   Connessione socket al server
   Parametri in ingresso:
          socket:   puntatore socket
          server:   indice selezione server
   Parametri in uscita:
          flag connessione eseguita
   -------------------------------------------------------------*/

   //Selezione indirizzo IP socket server
   $host = ($server == 1) ? 'XXX.XXX.XXX.XXX' : 'YYY.YYY.YYY.YYY';
   //Porta socket server
   $port = XXXX;
   //Timeout connessione nel caso di rifiuto del primo tentativo [ms]
```

```
   $timeout = 4000;

   socket_set_block($socket);
   //Se connessione non riuscita
   if (!socket_connect($socket, $host, $port))
   {
      //Con codice errore: ECONNREFUSED - connessione rifiutata
      if (socket_last_error($socket) === 111)
      {
         socket_set_nonblock($socket);
         $time = microtime(true);
         //Iterazione tentativi di connessione
         while (!@socket_connect($socket, $host, $port))
         {
            //Con codice errore: EISCONN - socket già connesso
            if (socket_last_error($socket) === 56)
               break;
            if ((microtime(true) - $time) > ($timeout * 1000))
            {
               socket_close($socket);
               return false;
            }
            //Pausa tra i tentativi
            usleep(250000);
         }
         socket_set_block($socket);
      }
      else
      {
         socket_close($socket);
         return false;
      }
   }
   //Connessione eseguita
   return true;
}
```

ed i metodi per la codifica degli errori:

```
function socketError()
{
   /*-------------------------------------------------------
   Codifica errore socket in stringa JSON
```

```
   Parametri in uscita:
          oggetto JSON con dati errore
   ------------------------------------------------------------*/

   return json_encode(array('error'   => true,
                            'source'  => 'Socket',
                            'code'    => socket_last_error(),
                            'message' => socket_strerror(
                                         socket_last_error())));
}

function arduinoError($code)
{
   /*------------------------------------------------------------
   Codifica errore Arduino in stringa JSON
   Parametri in uscita:
          oggetto JSON con dati errore
   ------------------------------------------------------------*/

   switch ($code)
   {
      case 100: $descr = 'sintassi comando non  valida'     ; break;
      case 101: $descr = 'codice comando non riconosciuto'  ; break;
      case 102: $descr = 'parametri aggiuntivi non corretti'; break;
      case 200: $descr = 'sintassi riposta non valida'      ; break;
      default:  $descr = '---'                              ; break;
   }
   return json_encode(array('error'   => true,
                            'source'  => 'Arduino',
                            'code'    => $code,
                            'message' => $descr));
}
```

3.3 Il problema "rompicapo"

Riporto un problema che mi ha fatto penare e che non avevo intercettato nelle sessioni di debug, ma solo successivamente: dopo 5-6 ore di utilizzo, inspiegabilmente, Arduino non rispondeva più alle invocazioni da parte del web server, eppure il micro era attivo, "vivo e vegeto".
Spegnendo e riaccendendo, tutto riprendeva a funzionare correttamente, per qualche ora, questo tempo sembrava dipendente da "quanto" si utilizzava la comunicazione.

Se si prova a fare una ricerca di problemi simili ("freeze" dell'Ethernet Shield) il forum di Arduino restituisce una montagna di segnalazioni, dovute alle più disparate cause, da chi non imposta

correttamente i pin CS (Chip Select) dell'interfaccia SPI, a chi ha probabilmente disturbi sull'alimentazione, fino a chi pilota periodicamente un segnale di reset alla scheda..

Fortunatamente mi sono imbattuto in alcune discussioni dove si sosteneva che sporadicamente la connessione veniva "chiusa male" lato client (sembra un fenomeno amplificato con alcuni browser come Chrome, che tra l'altro uso sullo smartphone) per cui, con la gestione della libreria dell'IDE di Arduino, uno dei socket del chip WizNET rimane indefinitamente nello stato "connesso" o "attesa chiusura" e non più utilizzabile.
A seconda del chip usato (W5100, W5200, W5500) sono disponibili 4 o 8 socket, per cui se l'evento si ripete prima o poi porta a bloccare tutti i canali disponibili rendendo impossibile ulteriori comunicazioni.
La soluzione proposta dall'utente SurferTim del forum di Arduino è semplice: inserire un timeout per i vari socket disponibili che ne forzi la chiusura in caso di mantenimento "anomalo" della connessione (superiore a 30 secondi), ed è quello che fa il metodo *checkSockStatus()* che ha inserito nel suo esempio di Arduino web server, qui lo vedete leggermente modificato per adattarlo alla mia versione di librerie:

```
void WebInterface::checkSockStatus()
{
   unsigned long thisTime = millis();
   SPI.beginTransaction(SPI_ETHERNET_SETTINGS);
   for (byte i = 0; i < MAX_SOCK_NUM; i++)
   {
      uint8_t s = W5100.readSnSR(i);
      if ((s == SnSR::ESTABLISHED) || (s == SnSR::CLOSE_WAIT))
      {
         if (thisTime - _connectTime[i] > 30000UL)
            W5100.execCmdSn(i, Sock_CLOSE);
      }
      else
         _connectTime[i] = thisTime;
   }
   SPI.endTransaction();
}
```

questa modifica si è dimostrata risolutiva nel mio caso.

4. Pagine web

Come server web ho utilizzato Apache, che si installa su Raspberry molto velocemente. Le pagine web hanno una struttura semplice, i file html contengono la disposizione dei controlli grafici, con l'assegnazione degli ID e delle classi per identificarli, lo stile è gestito nei file css, vi è un file principale (main.css) e poi ogni pagina ha in cascata il proprio, ogni pagina ha inoltre il proprio file

js contente il codice javascript.

Riporto un esempio per una pagina base contenente due pulsanti (cmdOn/cmdOff) per comandare un dispositivo ed un led (cmdState) che ne indica lo stato. Il codice HTML è:

```
<!DOCTYPE html><html>
  <head>
    <title>Titolo pagina</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <script                type="text/javascript" src="js/jquery/jquery.min.js"></scrip
    <script                type="text/javascript" src="js/socket.js"></script>
    <script                type="text/javascript" src="js/idle.js"></script>
    <script                type="text/javascript" src="js/example.js"></script>
    <link rel="stylesheet" type="text/css" href="css/main.css"/>
    <link rel="stylesheet" type="text/css" href="css/example.css"/>
  </head>
  <body onselectstart="return false" ondragstart="return false">
    <h1>ESEMPIO</h1>
    <fieldset>
      <legend>Comando ad Arduino</legend>
      <table>
        <tr>
          <td><input type="image" class="button" id="cmdOn"  src="images/on.png"  disab
          <td><input type="image" class="button" id="cmdOff" src="images/off.png" disab
          <td/>
          <td><img id="cmdState"/></td>
        </tr>
      </table>
    </fieldset>
  </body>
</html>
```

ipotizzando che lato Arduino (server con indice 0) siano gestiti i tre comandi:

- 800: comando di spegnimento
- 801: comando di accensione
- 810: interrogazione dello stato

e che questi tre comandi abbiano tutti come parametro di ritorno lo stato del dispositivo (0/1), il codice JavaScript (example.js) sarebbe:

```
$(document).ready(function()
{
  socketLoop(5000, cmdStateView, 0, 810);
```

```
  $("#cmdOn").click(function()
  {
    socket(cmdStateView, 0, 801);
  });
  $("#cmdOff").click(function()
  {
    socket(cmdStateView, 0, 800);
  });
});
function cmdStateView(data)
{
  var cmdState = (parseInt(data[0]) == 1);

  $("#cmdOn").attr( "disabled",  cmdState);
  $("#cmdOff").attr("disabled", !cmdState);
  $("#cmdState").attr("src", "images/led." + (cmdState ? "red" : "off") + ".png");
}
```

dove abbiamo anche un refresh ogni 5 secondi in modo da aggiornare lo stato del dispositivo se questo venisse comandato da un altro utente o in automatico.
La pagina visualizzata con il codice sopra riportato è del tipo:

Figura 6 - Pagina di esempio ON/OFF

4.1 Tastiera virtuale

Prevedendo di utilizzare l'interfaccia sul pannello touch del Raspberry, era necessaria una tastiera virtuale, da visualizzare automaticamente quando si clicca su una casella editabile.
Io ho utilizzato questa (secondo me ottima) jQuery OSK, che ha anche il layout personalizzabile, ad esempio nel caso di caselle che prevedono un input numerico ho adottato la seguente impostazione:

```
$(obj).keyboard(
{
   layout:             "custom",
   customLayout:   {   "default":
                   [   "{del} {left} {right} {b}",
                       "7 8 9 /",
                       "4 5 6 +",
                       "1 2 3 -",
                       "0 . {c} {a}"  ]}
});
```

è poi possibile abilitare o meno il copia/incolla, abilitare il tasto "Accept" solo con input valido, filtrare in input solo i tasti visualizzati, personalizzare il metodo di validazione usando le espressioni regolari e molto altro, un esempio:

```
$(obj).keyboard(
   restrictInput:      true,
   preventPaste:       false,
   autoAccept:         false,
   acceptValid:        true,
   validate:           function(keyboard, value, isClosing)
   {
      var chk;            //Flag di dato valido
      switch (dataType)
      {
         case "hour":     //Dato "ora" 0÷23
            chk = /^(\d$|[0-1]\d$|2[0-3]$)/.test(value);
            break;
         case "minute":          //Dato "minuti" 0÷59
            chk = /^(\d$|[0-5]\d$)/.test(value);
            break;
         case "byte":     //Dato "byte" 0÷255
            chk = /^\d{1,3}$/.test(value) && parseInt(value) < 256;
            break;
         default:
            chk = true;
```

```
        }
        if (!chk && isClosing)
           alert("Inserire un dato valido");

        return chk;
    }
});
```

ecco due screenshot di come ho gestito la tastiera per input numerici o testuali:

Figura 7 - Tastierino numerico

Figura 8 - Tastiera

4.2 Ritardo di inattività

Pensando soprattutto all'utilizzo con il pannello, quindi con l'ultima pagina visualizzata che rimane attiva anche per ore, magari con comandi di aggiornamento ogni pochi secondi, ho pensato di inserire un timer di inattività che commuti in automatico su una pagina "stupida" (logo.html) che non contenga richieste automatiche al server.
A questo scopo tutte le pagine richiamano il file *idle.js* che contiene il seguente codice:

```
var idleTimer = null;                    //Id timer monitoraggio inattività

$(document).ready(function()
{
  //Monitoraggio eventi che interrompono il periodo di inattività utente
  $("*").on("mousemove keydown scroll", idleStart);
  idleStart();                        //All'entrata nel documento inizia il periodo di i
});
function idleStart()
{
  clearTimeout(idleTimer);             //Cancellazione eventuale timer istanziato in pr

  idleTimer = setTimeout(function () //Istanza timer inattività utente
  {
    $("#main", window.parent.document).attr("src", "logo.html");
  },
```

```
  1500000);
}
```

la pagina visualizzata dopo 25 minuti di inattività è questa, da dove è anche possibile spegnere il Raspberry:

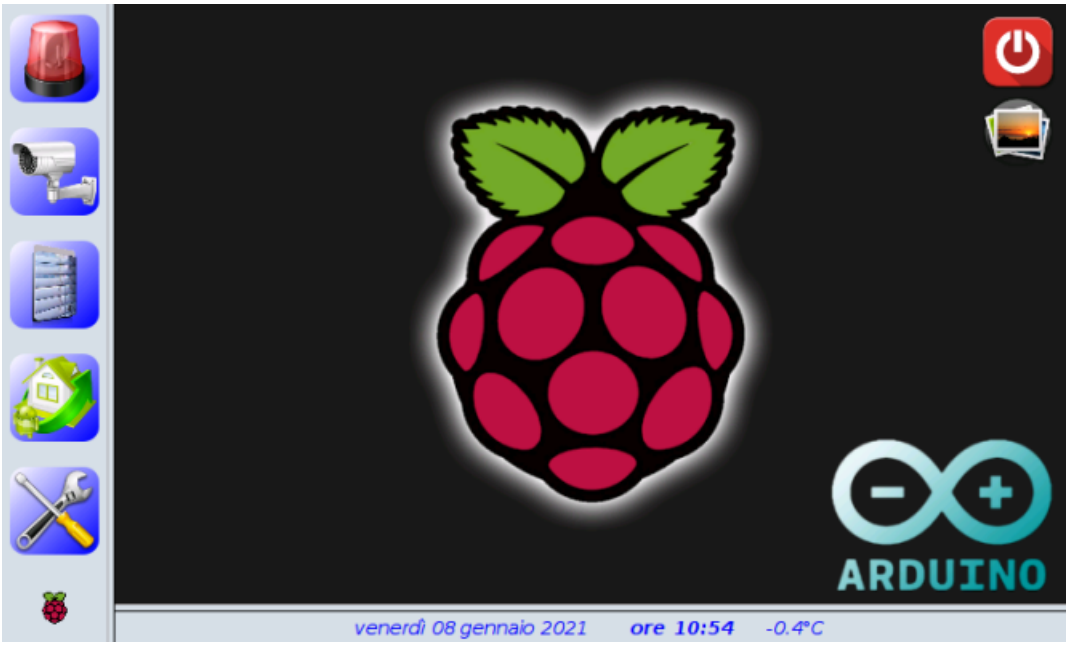

Figura 9 - Pagina di inattività

4.3 Accesso da remoto

Prima di rendere accessibile da remoto il webserver (Raspberry) mediante port-forwarding, ci sono montagne di guide in rete da seguire sulle precauzioni da prendere, una su tutte la modifica della password di default dell'utente pi, o meglio, creare un nuovo utente con password sicura e cancellare l'utente di default.

Lato webserver, per avere un minimo di sicurezza, ho deciso di rendere necessario l'inserimento di una password quando si accede da remoto, ma non dalla rete locale. Per fare questo ho inserito il file .htaccess nella cartella principale, con le seguenti direttive:

```
# Richiesto accesso per connessioni remote
<If "%{HTTP_HOST} == 'localhost'">
   Require all granted
</If>
<ElseIf "-R '192.168.0.0/24'">
   Require all granted
</ElseIf>
<Else>
```

```
    AuthType Basic
    AuthName "Richiesto accesso"
    AuthUserFile /var/www/html/pwd/.htpasswd
    require valid-user
</Else>

# Vietata navigazione cartelle
<IfModule mod_autoindex.c>
  Options -Indexes
</IfModule>
```

il file .htpasswd può essere creato con uno script o più semplicemente, dalla shell con il comando:

```
htpasswd -c .htpasswd <nome utente>
```

verrà chiesta la password ad associare all'utente, che sarà salvata nel file criptandola.
Per aggiungere utenti al file il comando è:

```
htpasswd .htpasswd <nome utente>
```

4.4 Screenshots

Riporto alcuni screenshots delle pagine realizzate:

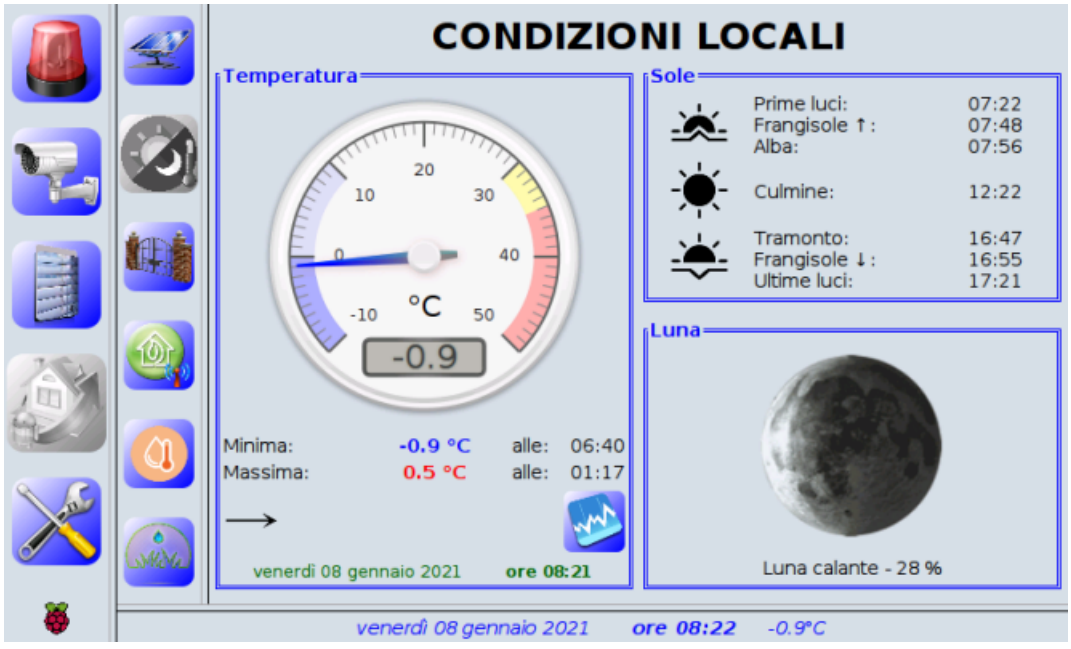

Figura 10-1 - Screenshot 1

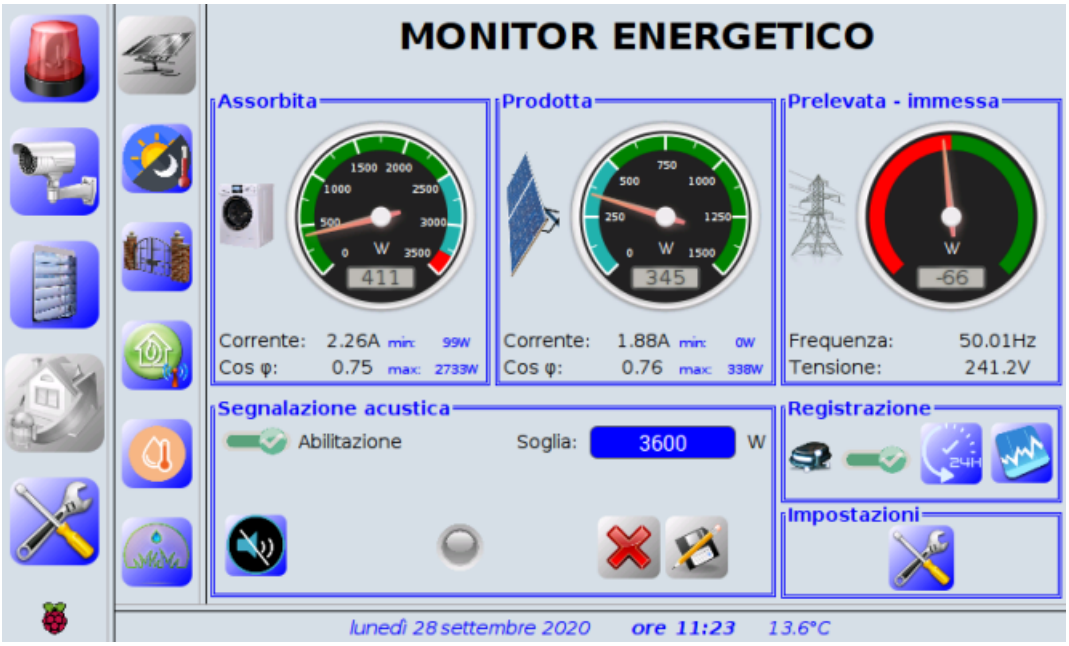

Figura 10-2 - Screenshot 2

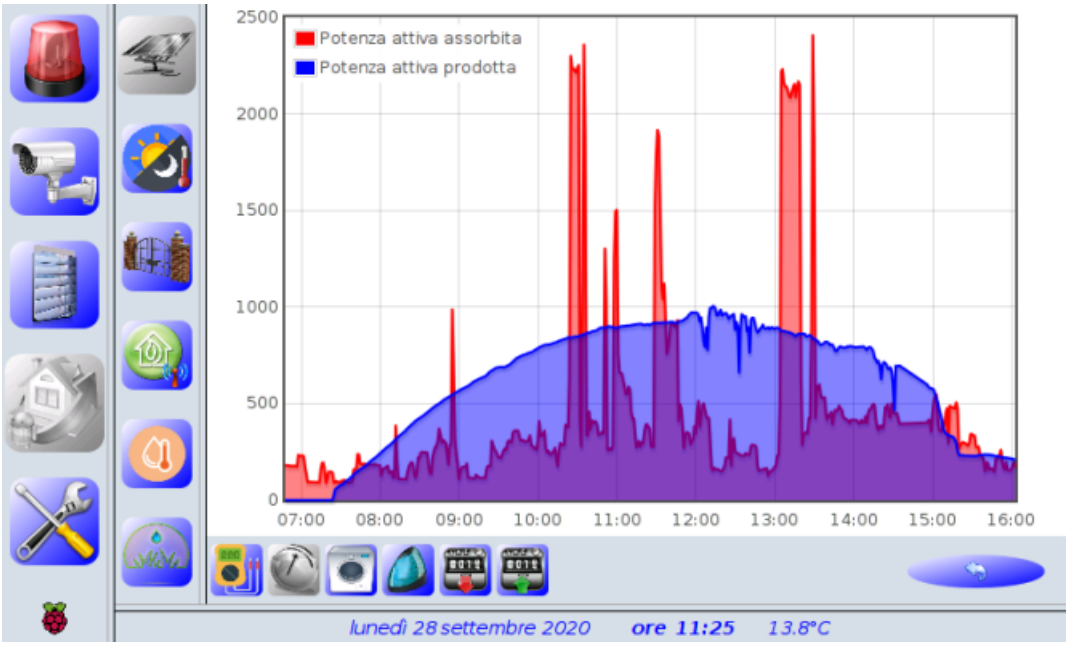

Figura 10-3 - Screenshot 3

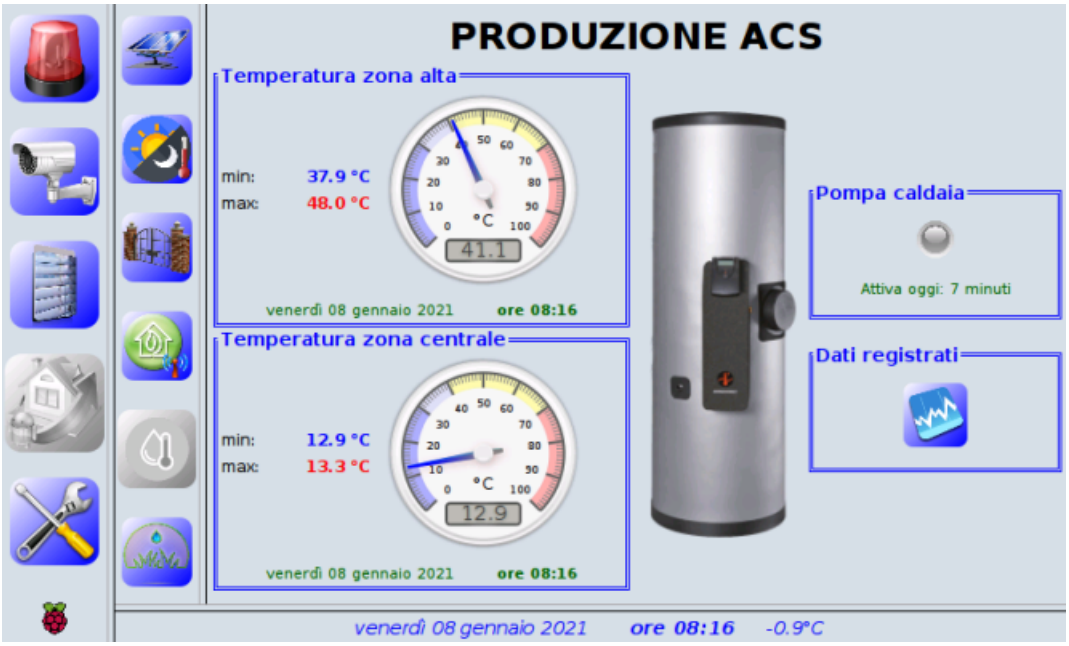

Figura 10-4 - Screenshot 4

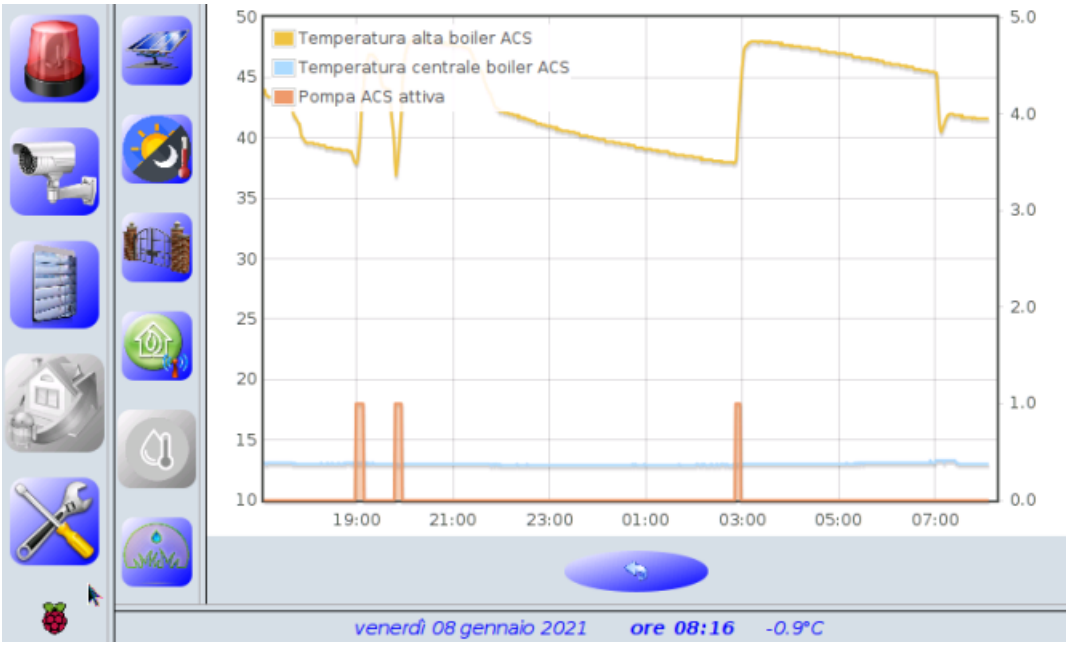

Figura 10-5 - Screenshot 5

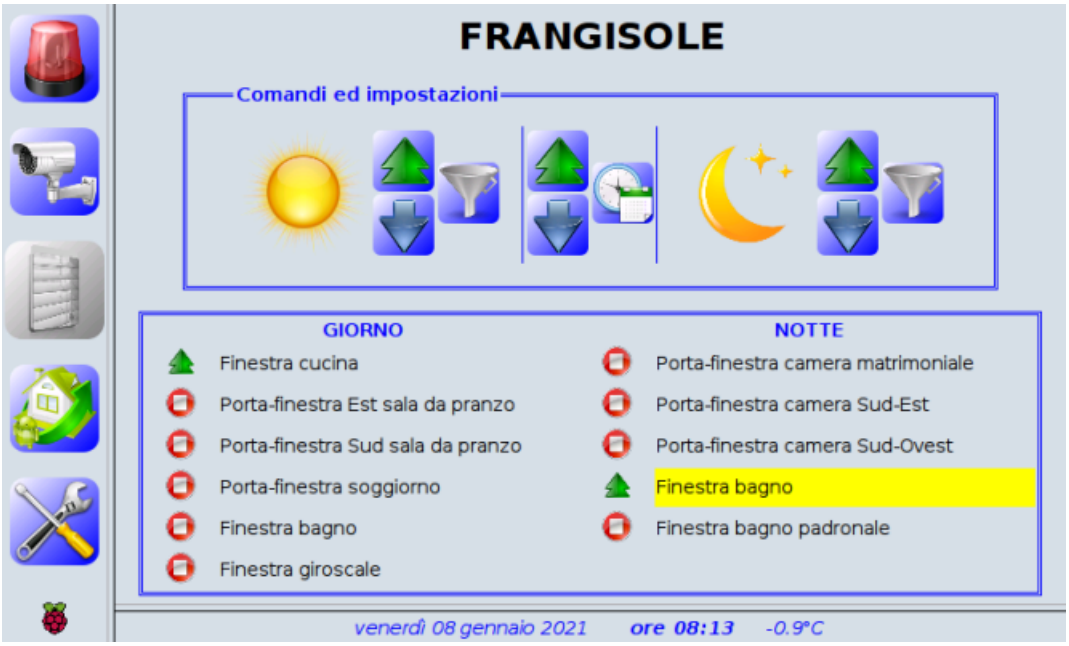

Figura 10-6 - Screenshot 6

Figura 10-7 - Screenshot 7

Figura 10-8 - Screenshot 8

Per gli indicatori a lancetta ho utilizzato la libreria JS Canvas Gauges, i grafici visualizzati sono quelli relativi ai log di EmonCMS (installato direttamente su Raspberry), mentre per il calcolo di alba/tramonto e le posizioni lunari ho usato la libreria JS SunCalc.

5. Interfaccia sul pannello

5.1 Script di avvio & browser

Per visualizzare l'interfaccia direttamente su Raspberry era necessario un browser leggero e veloce, dopo averne provato alcuni ho trovato kweb (Minimal Kiosk Browser) che a mio avviso funziona davvero bene, rendendo sufficientemente "fluida" la navigazione.
Ho inserito uno script (*boot.sh*) lanciato all'avvio del Raspberry, modificando il file:

```
~/.config/lxsession/LXDE-<nome utente>/autostart
```

che quindi contiene:

```
@lxpanel --profile LXDE-<nome utente>
@pcmanfm --desktop --profile LXDE-<nome utente>
@boot.sh
```

lo script *boot.sh* a sua volta fa partire vari script che vedremo in seguito, uno di questi avvia kweb visualizzando l'indirizzo `http://localhost`.

5.2 Screensaver-cornice digitale

Era carino, dopo un periodo di inattività, avviare sul pannello una slide-show con delle foto, che però venisse interrotta con gli eventi di touch del pannello (come uno screen-saver), per questo ho preso spunto dal forum Raspberry.
Innazitutto serve installare:

- *feh*: programma leggero per visualizzazione immagini e slideshow
- *xprintidle*: utility per controllare il tempo di inattività utente
- *xbindkeys*: utility per associare comandi della shell a determinati tasti della tastiera o mouse

```
sudo apt-get install feh

sudo apt-get install xprintidle

sudo apt-get install xbindkeys
```

Ho poi inserito in `~/bin` il seguente script (*screensaver.sh*), che accetta come parametro il ritardo di inattività in secondi per avviare lo screensaver (default un minuto):

```
#!/bin/bash
#Script per screensaver (ritardo [s])

if [ -z "$1" ]; then
   delay=60000
else
   delay=$(($1*1000))
fi



cd ~/Photos/Screensaver
while sleep $((1)); do
   idle=$(xprintidle)
   if [ $idle -ge $delay ]; then
      xbindkeys -n -f ~/bin/xbindkeys.exit.feh &
      feh -x -F -Z -Y -r -z -q -A slideshow -D 6
   fi
done
```

il comando **feh** con i vari parametri avvia la presentazione, cambiando foto ogni 6 secondi, mentre il comando **xbindkeys** consente la chiusura di feh con un click, per fare questo è necessario anche il file **xbindkeys.exit.feh**, che contiene l'associazione al tasto sinistro del mouse dei comandi di terminazione di feh e dello stesso xbindkeys:

```
"pkill -n feh; pkill -n xbindkeys"
b:1
```

infine ho lanciato screensaver.sh da boot.sh, ovvero all'avvio del Raspberry.

5.3 Visualizzazione telecamera

Altra cosa: ho una telecamera IP con risoluzione 1280x720 che inquadra la zona dell'ingresso di casa. Quello che volevo era poter visualizzare il flusso video di questa telecamera direttamente dall'interfaccia sul pannello, entrando in una specifica pagina ed anche che, al suono del campanello, questa venisse visualizzata in automatico.

Su Raspberry Pi 2 con Raspbian, uno dei pochi modi per poter utilizzare l'accelerazione HW della scheda video è con omxplayer, un media player pre-installato.
Il player viene semplicemente lanciato da riga di comando ed è possibile impostare la posizione ed i limiti della finestra di visualizzazione, che va in sovrapposizione a tutto il resto (va direttamente all'hardware video).

A questo scopo ho predisposto la seguente funzione js, che prevede in ingresso un parametro da trasmettere allo script camera.php lato server, indicante la richiesta di avvio (play) o chiusura (stop) della visualizzazione, il tutto avviene tramite chiamata asincrona:

```
function cameraView(command)
{
   $.ajax(
   {
      url:        "php/camera.php",
      data:       {cmd: command},
      timeout:    "5000"
   });
}
```

Lo script lato server è il seguente, ovviamente accetta il comando solo se proviene dal pannello connesso direttamente al Raspberry:

```
<?php
   if ($_SERVER['REMOTE_ADDR'] == '::1')
   {
      if ($_GET['cmd'] == 'play')
         shell_exec('omxplayer -n -1 --live --fps 15 --win 86,1,799,450
                     rtsp://<user>:<pwd>@xxx.xxx.xxx.xxx/live1.264
                     > /dev/null 2>/dev/null &');
      else
         shell_exec('pkill -n omxplayer > /dev/null 2>/dev/null &');
```

```
    }
?>
```

A questo punto ho la possibilità di visualizzare la telecamera, come fosse una pagina qualunque dell'interfaccia, manca l'avvio automatico al suono del campanello.
Il mio citofono (Urmet) ha la possibilità di pilotare una suoneria supplementare o un relè ripetitore di chiamata tramite i morsetti S+ e S-, che quindi ho utilizzato per "chiudere" un ingresso GPIO sul Raspberry:

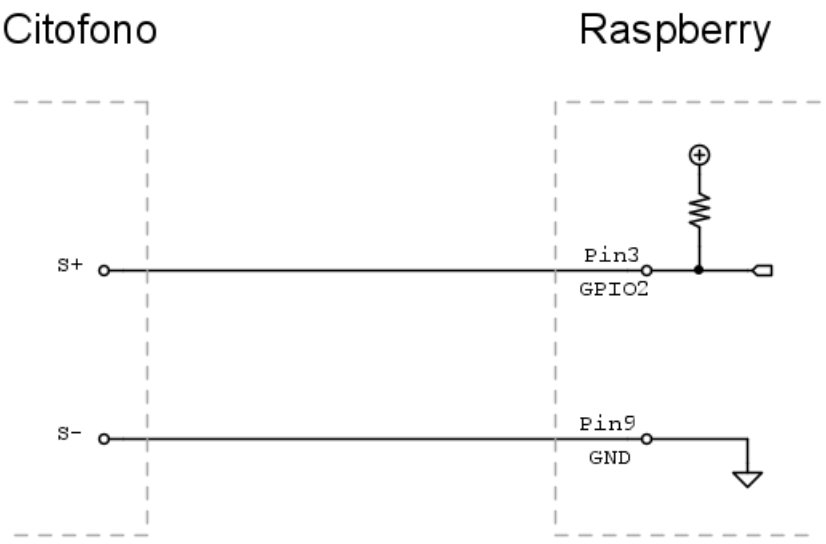

Figura 11 - Attivazione GPIO con campanello

La prima idea che mi è venuta in mente è quella di effettuare, con l'attivazione del GPIO, un "click virtuale" sul pulsante di visualizzazione della telecamera, pulsante che rimane sempre nella stessa posizione, nella "menù bar" a sinistra, ho quindi installato l'utility *xdotool* che consente di emulare movimenti e click del mouse:

```
sudo apt-get install xdotool
```

infine, all'avvio del Raspberry, ovvero da boot.sh, ho richiamato il seguente script Phyton, che sul fronte di discesa del GPIO2 sposta il puntatore del mouse sul pulsante voluto e poi emula un click del pulsante sinistro:

```
!/bin/python
#Script di commutazione pagina al suono del campanello

# Importazione moduli
import RPi.GPIO as GPIO
import time
```

```
import os

# Pin utilizzato: GPIO2 (pin 3)
GPCHANNEL = 2
# Impostazione pin di ingresso con pull-up interno
GPIO.setmode(GPIO.BCM)
GPIO.setup(GPCHANNEL, GPIO.IN, pull_up_down = GPIO.PUD_UP)

# Metodo di callback (eseguito dall'interrupt)
def Bell(channel):
   os.system("xdotool mousemove --sync 40 120")
   time.sleep(1)
   os.system("xdotool click 1")

# Gestione evento di fronte negativo sull'ingresso monitorato
GPIO.add_event_detect(GPCHANNEL, GPIO.FALLING, callback = Bell)

# Loop per mantenere script attivo
while 1:
   time.sleep(1)
```

Il limite di questa gestione è che la visualizzazione della telecamera, fatta sfruttando l'accelerazione HW della scheda video, non è usufruibile tramite accesso remoto (esempio VNC) né tantomento da webserver su altri dispositivi, ma solamente dal pannello connesso direttamente al Raspberry. E' però un limite accettabile, visto che la telecamera IP è accessibile direttamente dalla rete e quindi può essere visualizzata tramite altre applicazioni (esempio VLC).

6. Auto-spegnimento con black-out

In rete riportano problemi di corruzione della scheda SD spegnendo brutalmente il Raspberry, nel mio caso con interruzione di tensione il Raspberry rimane alimentato assieme all'Arduino dalla batteria tampone dell'alimentatore.
Mantenere acceso Rasberry e monitor accorcia però inutilmente la durata di alimentazione tramite batteria. In caso di black-out superiore a qualche minuto ho pensato quindi di inviare un segnale da Arduino per invocare lo spegnimento del Raspberry, così facendo anche il monitor andrebbe in standby riducendo drasticamente il consumo.

Il meccanismo usato è lo stesso per la visualizzazione della telecamera: uno script python avviato all'accensione che testa il fronte di un GPIO per comandare lo spegnimento.

La connessione per attivare l'ingresso è:

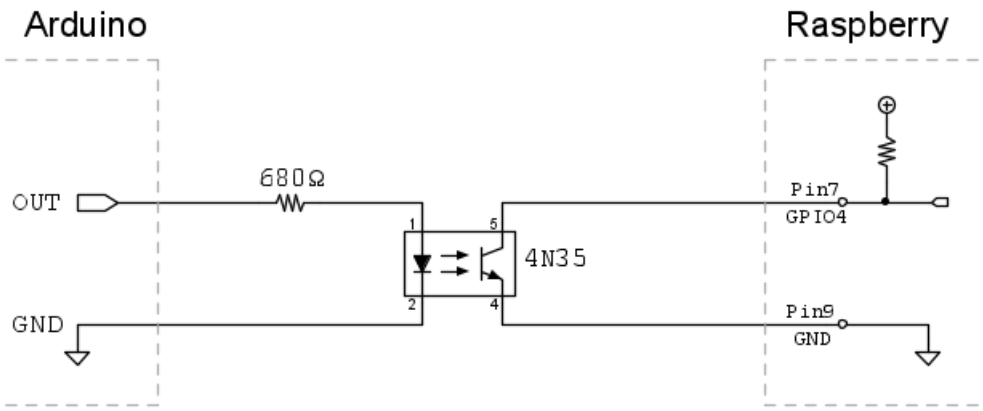

Figura 12 - Comando di spegnimento

Mentre lo script che monitora l'ingresso è il seguente, c'è una iterazione poco elegante per evitare spegnimenti intempestivi:

```
#!/bin/python
#Script di spegnimento sistema da GPIO

# Importazione moduli
import RPi.GPIO as GPIO
import time
import os

# Pin utilizzato: GPIO4 (pin 7)
GPCHANNEL = 4
# Impostazione pin di ingresso con pull-up interno
GPIO.setmode(GPIO.BCM)
GPIO.setup(GPCHANNEL, GPIO.IN, pull_up_down = GPIO.PUD_UP)

# Metodo di callback (eseguito dall'interrupt)
def Shutdown(channel):
   for n in range(5):
      time.sleep(0.01)
      if GPIO.input(channel) != GPIO.LOW:
         time.sleep(1)
         return
   os.system("sudo shutdown -h now")

# Gestione evento di fronte negativo sull'ingresso monitorato
GPIO.add_event_detect(GPCHANNEL, GPIO.FALLING, callback = Shutdown)
```

```
# Loop per mantenere scrip attivo
while 1:
   time.sleep(1)
```

A questo punto, per riavviare il Raspberry a black-out terminato, è sufficiente portare a massa il pin 5:

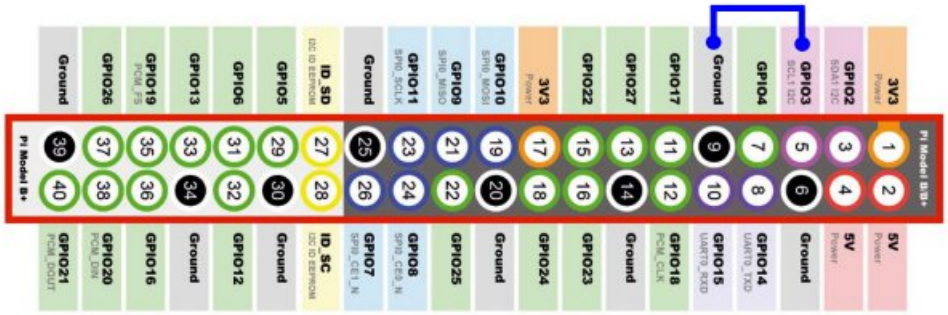

Figura 13 - Riavvio

cosa che attualmente ho fatto tramite un pulsante libero (a contatto pulito) del videocitofono a fianco. In futuro userò un segnale da Arduino per avere un riavvio da remoto o automatico quando torna l'alimentazione.

7. Messa in servizio del display

Per la messa in servizio all'epoca ho installato Raspbian (ora chiamato Raspberry Pi OS) Jessie, con desktop. Per le varie operazioni, non essendo minimamente esperto, ho seguito le guide disponibili online, guide che sono obsolete con la versione attuale, per cui le ho lasciate sui miei appunti. Riporto solo la procedura per la messa in servizio del pannellino che potrebbe essere utile:

- editare il file config:

```
sudo nano /boot/config.txt
```

con le seguenti impostazioni:

```
#remove black borders
disable_overscan=1
#force hotplug
hdmi_force_hotplug=1

#set CVT as default
hdmi_group=2
hdmi_mode=87
#set specific CVT mode (800x480 60Hz 15:9)
```

```
hdmi_cvt 800 480 60 6 0 0 0
#increase HDMI signal strength
config_hdmi_boost=4
```

- salvare con <Ctrl+O> ed uscire con <Ctrl+X>
- per la configurazione del touch, installare e lanciare il programmino *evtest*:

```
sudo apt-get install evtest -y
```

```
evtest
```

- selezionare gli eventi corrispondenti a "eGalax Inc. USB TouchController"
- toccare con una penna gli angoli *alto-sinistra* e *basso-destra* annotando i valori **ABS_X** e **ABS_Y**
- uscire con <Ctrl+C>
- editare il file 10-evdev.conf:

```
sudo nano /usr/share/X11/xorg.conf.d/10-evdev.conf
```

scorrere fino in fondo ed aggiungere prima di EndSection:

```
Option "SwapAxes" "1"
Option "InvertX" "1"
Option "Calibration" "<min x> <max x> <min y> <max y>"
```

gli assi del touch sono invertiti, nel valore <min x> è da scrivere il valore minimo di ABS_Y annotato in precedenza e così via.. nel mio caso ho: `Option "Calibration" "86 1970 230 1880"`

- salvare con <Ctrl+O> ed uscire con <Ctrl+X>
- riavviare:

```
sudo reboot
```

8. Conclusione

Alla fin fine è stata un'esperienza molto interessante, per me una buona occasione di apprendimento ed approfondimento, sia per giocare con l'ennesimo linguaggio che per riscoprire un po' di elettronica elementare (campo dove, come dice qualcuno qui nel forum, non "capisco una mazza"). Il risultato ottenuto è superiore alle aspettative iniziali, quindi sono soddisfatto. Certo il Raspberry non è un "fulmine di guerra", ma la navigazione tra le pagine è buona, se si accede da smartphone risulta ancora più veloce.

Mi rendo conto che l'articolo si è allungato parecchio, diventando un polpettone, meglio concludere: vedrò di migliorare in un eventuale prossimo articolo.

9. Codice

Il codice per la comunicazione illustrato in questo articolo è scaricabile qui.

10. Riferimenti

Visual Micro - plugin per Visual Studio
PlatformIO - plugin per Visual Studio Code
OpenEnergyMonitor - energy monitor con Arduino
EmonCMS - processazione, logging e visualizzazione dati
PString - libreria manipolazione stringhe per Arduino
Forum Arduino - esempio web server di SurferTim
Mottie Keyboard - tastiera virtuale con jQuery
Canvas Gauges - libreria JS indicatori vari
SunCalc - libreria JS calcolo alba/tramonto e posizioni lunari
Forum Raspberry - screensaver slideshow
Omxplayer - media player da riga di comando