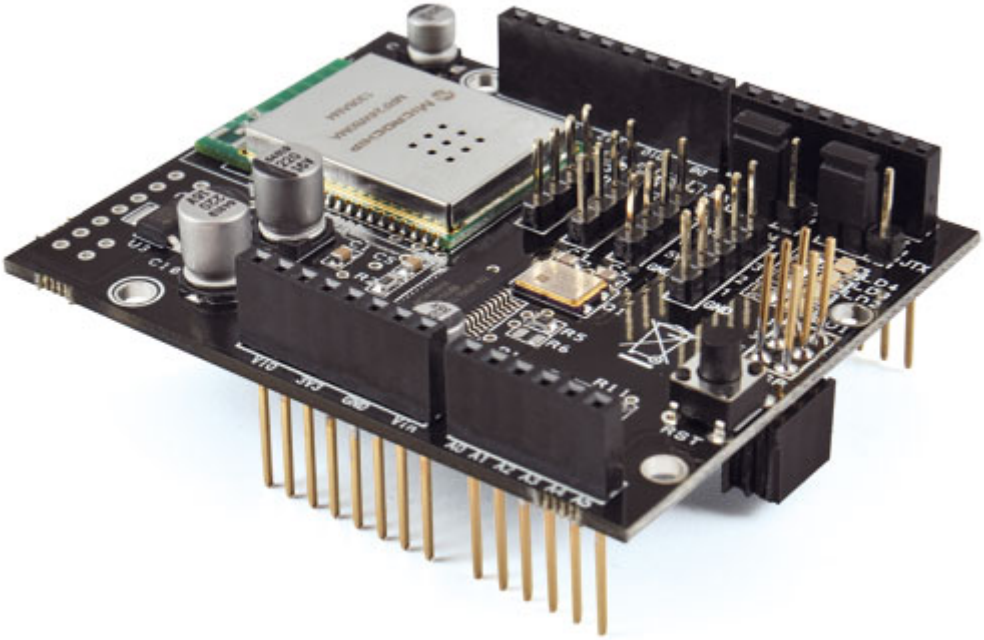
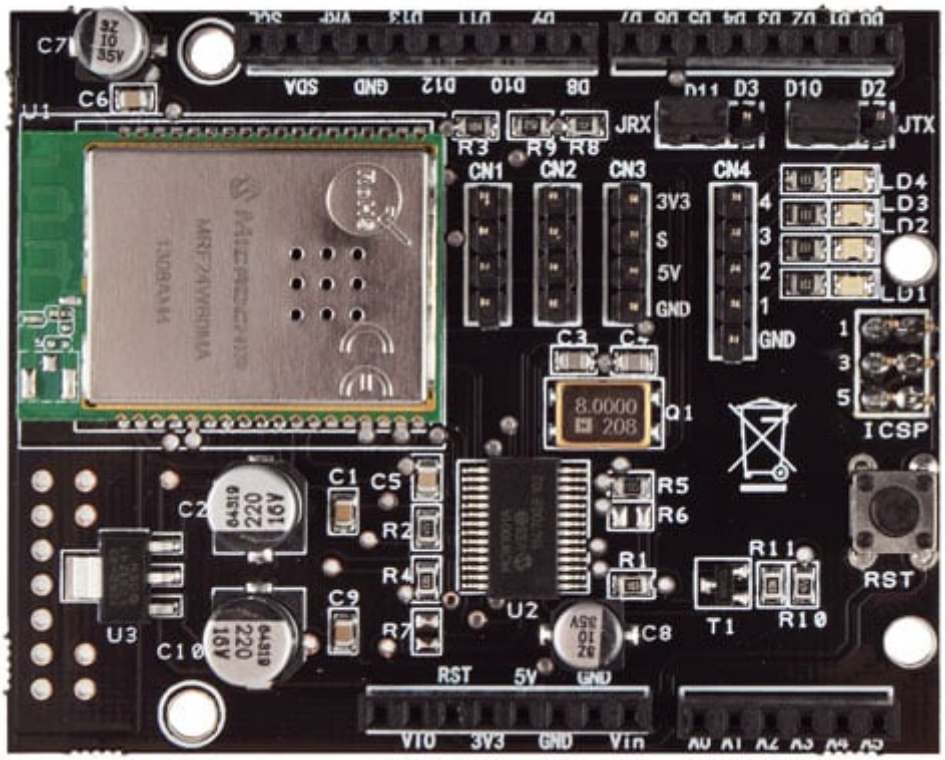
Shield WiFi per Arduino

Prezzo: 45.90 €

Tasse: 10.10 €

Prezzo totale (con tasse): 56.00 €





Shield Wi-Fi per Arduino basata sul modulo transceiver (MRF24WB0MA) Wi-Fi a 2,4 GHz standard 802.11 IEEE della Microchip. Il modulo MRF24WB0MA dispone di un'antenna integrata sufficiente alla copertura di qualche decina di metri. Lo shield Wi-Fi utilizza una versione personalizzata della SoftwareSerial, perché la comunicazione con il controller della scheda avviene tramite collegamento seriale. Per questo motivo sono stati usati i pin digitali 2 e 3 (che quindi non possono essere utilizzati per altri scopi) ; inoltre è utilizzato il pin digitale 7 per il reset della scheda. I pin 2 e 3 possono essere scambiati con 10 e 11 mediante ponticelli, ma vanno anche ridefiniti nella libreria (file MWiFi.h). Inoltre la scheda ha 4 LED di cui due sono utilizzati dalla libreria. Più precisamente: il LED 0 è attivato all'accensione e start-up della scheda mentre il LED 1 viene acceso quando si stabilisce la connessione con la rete. Gli altri due possono essere utilizzati dall'utente.
Arduino, come si sa, è composto da un hardware minimale, mentre i collegamenti con la rete Internet richiedono una gestione asincrona, e sostanzialmente multi-task, dei protocolli di comunicazione. Per questo motivo sulla scheda proposta è stato previsto un gestore dello stack TCP/IP, in modo da liberare Arduino da alcune incombenze di base.

La libreria MWiFi

La libreria MWiFi va posizionata (dopo la decompressione) nella directory “libraries” dell' Ide di Arduino, come le altre librerie. Va utilizzata includendo negli sketch il file <MwiFi.h>. A questo punto va istanziata come oggetto e può essere utilizzata. La prima funzione da richiamare è begin() che serve a inizializzare la scheda:

```
#include <MWiFi.h>
MWiFi WIFI;
void setup()
{
WIFI.begin();
:
```

Lo start-up della scheda provoca l'accensione del primo LED. Proviamo, ora, a collegare Arduino in rete utilizzando un access-point che potrebbe essere quello del router Wi-Fi casalingo per esempio con nome (Ssid) : “D-Link-casa”.

```
WIFI.ConnSetOpen(“D-Link-casa”);
```

nel caso la rete fosse non protetta; oppure:

```
WIFI.ConnSetWPA(“D-Link-casa”,”miarete”);
```

nel caso la rete avesse protezione Wpa con password “miarete”. Le precedenti funzioni servono a predisporre la scheda alla connessione; ma la vera e propria connessione si ha chiamando la funzione:

```
WIFI.Connect();
```

Se la connessione avviene si accende il secondo LED. Attenzione! Nel caso di rete protetta la connessione può richiedere anche più di mezzo minuto, perché il controller della scheda deve codificare la chiave tramite la password. L'attuale versione della libreria prevede un reset automatico di Arduino nel caso la connessione venga persa. Anche errori rilevati dal controller provocano un reset automatico. In questo modo un eventuale blocco è scongiurato ed il sistema può essere “unattended”.
Sono state aggiunte anche funzioni dirette di connessione, per comodità. Queste funzioni preparano la connessione e la eseguono in un unico step. Inoltre è stata aggiunta la possibilità di generare la chiave numerica dalla password in modo da utilizzare successivamente questa al posto della password. Infatti l'accesso con password, prevedendo ogni volta l'elaborazione della chiave numerica, può richiedere anche un minuto, mentre l'accesso con chiave è velocissimo.
Ad Arduino, viene assegnato, dal router, un indirizzo IP dinamico, perché il comportamento di default della scheda è questo; ma volendo si può imporre un indirizzo fisso. L'indirizzo assegnato può essere richiesto da una funzione della libreria.
A questo punto scegliamo di far fare ad Arduino la funzione di server. Per prima cosa chiamiamo la funzione openServerTCP(), che crea l'ascoltatore su una certa porta (per esempio 5000), e poi mettiamo in loop la ricezione di una eventuale richiesta di link:

```
int ssocket=WIFI.openServerTCP(5000);
void loop()
{
int csocket=WIFI.pollingAccept(ssocket);
:
```

Le variabili intere ssocket e csocket sono dei riferimenti (handle) rispettivamente al server e al socket di collegamento con l'eventuale computer che vuole effettuare il link. La funzione pollingAccept() ritorna un numero minore di 255 se il collegamento è stato richiesto, o 255 se non c'è nessuna richiesta di collegamento in arrivo.
Nel caso il collegamento sia stato stabilito noi potremo spedire o ricevere messaggi facendo riferimento a questo csocket. Per esempio per ricevere un record , cioè una stringa terminante con un line-feed possiamo utilizzare la funzione:

```
char *line=WIFI.readDataLn(csocket);
```

Ci verrà restituita una “null-terminated string” ma senza il line-feed. La libreria in questo caso utilizza un buffer predefinito di 81 caratteri (ma la sua lunghezza può essere modificata nel suo define). Quindi non abbiamo bisogno di fornirlo noi. Sono comunque previste altre possibilità. Se invece vogliamo rispondere possiamo utilizzare la funzione:

```
WIFI.writeDataLn(csocket,answer);
```

La variabile che abbiamo chiamato answer corrisponde ad un buffer di char. Però deve essere una “null terminated string”. N.B. Quando viene utilizzata una “null-terminated string” vuol dire che non dobbiamo fornire la lunghezza dei caratteri utili nel array, perché la funzione la calcola automaticamente, potendo basarsi sul carattere terminale nullo. La maggior parte delle funzioni c gestisce e produce questo tipo di stringa da non confondere con l'oggetto String presente anche nel language reference di Arduino.
In questo semplice modo abbiamo stabilito e utilizzato un collegamento Wi-Fi con un computer remoto. Nella libreria, tra gli esempi riportarti, c'è un esempio di server chiamato CommandServer che permette di comandare Arduino utilizzando un programma tipo telnet sul computer remoto. Per semplificare i test è stato aggiunto un programma Java che opera come telnet.
Se invece volessimo far agire Arduino come un “client” che si collega ad un server, la situazione è ancora più semplice perché dobbiamo solo creare un socket di collegamento:

```
int csocket=WIFI.openSockTCP(“192.168.1.2”,5000);
```

E se csocket è valida (minore di 255), vuol dire che il collegamento, con il computer all'indirizzo 192.168.1.2 sulla porta 5000, è stato stabilito. A questo punto possiamo utilizzare le funzioni di scrittura e lettura viste precedentemente. Tra gli esempi ce n'è uno (SendData) che si collega ad un server remoto per spedirgli ad intervalli regolari letture di sensori. Ovviamente è necessario un programma server sul computer remoto. Per facilitare i test, insieme alla libreria è stato fornito un programma Java che riceve i dati e li scarica su un file aggiungendo un time-stamp (per ovviare alla mancanza di un Rtc su Arduino).
Oltre a queste funzionalità di base, nella libreria sono presenti tutte le funzioni necessarie per definire diversi parametri quali: l'indirizzo di mascheramento della rete (default 255.255.0.0), l'eventuale indirizzo di Gateway, la lettura del codice MAC della scheda ecc.
In particolare sono presenti delle funzioni per rilevare access-point presenti e visibili nell'ambiente. Per esempio per rilevare tutte le reti presenti:

```
int nn=WIFI.scanNets();
```

La variabile intera nn conterrà il numero di reti rilevate. Mentre la funzione:

```
char* net=WIFI.getNetScanned(i);
```

restituirà le caratteristiche (sotto forma di record) della rete rilevata. Infine, per comodità, è stata prevista una funzione che restituisce il nome della rete non protetta, più potente in termine di segnale.
Rimandiamo alla documentazione della libreria la descrizione di tutte le funzionalità disponibili. La libreria contiene un help (in inglese) ed è documentata nei file di codice (in particolare i file .h).
Ma la libreria non si esaurisce nella sola gestione di connessione e socket. Infatti è inclusa una classe derivata (e quindi specializzata) che gestisce il protocollo Http: il protocollo del Web.
Il protocollo HTTP è un protocollo che prevede una richiesta ed una risposta, sempre. Sia la richiesta che la risposta fanno viaggiare in rete, pacchetti formati da alcune intestazioni (header) e dati veri e propri (come le pagine Html, immagini, video o anche semplice testo).
Fig. 2 – Comunicazione HTTP
Per scaricare l'utente di tutta questa problematica, la libreria per HTTP, si preoccupa di formare questi pacchetti utilizzando, inoltre, in modo spinto la modalità PROGMEN; cioè la possibilità di porre in area flash di programma le costanti ed in particolare i testi. N.B. Il protocollo HTTP è un protocollo testuale: usa solo caratteri. L'uso della memoria flash per i testi permette di risparmiare l'esigua ram di Arduino.

La libreria HTTP

Essendo una classe derivata di MWiFi, essa eredità tutte le funzioni di MWiFi, ma se si vogliono utilizzare le nuove funzioni bisogna includere il file HTTPlib.h (al posto di MwiFi.h) ed istanziare un oggetto HTTP:

```
#include <HTTPlib.h>
HTTP WIFI;
```

Riguardo alla connessione con un access-point, ed alla gestione dei socket tutto rimane come prima (forse ora sceglieremo la porta 80); ma anche questa volta dobbiamo decidere se far fare ad Arduino le veci di un server (questa volta Web Server) o quelle di un client che accede ad un application web server (come Tomcat, GlassFish, Jboss, PHP ecc.).
Supponiamo che vogliamo realizzare un Web Server da interrogare con un qualunque browser. Per far funzionare Arduino come Web Server dobbiamo predisporre le risorse che lui può mettere a disposizione. Cioè le pagine html di risposta. Queste pagine saranno memorizzate in aree PROGMEM per i motivi detti sopra. Es:

```
prog_char pageindex[] PROGMEM=
"<html><head>"
"<title>Index</title>"
:
```

A questo punto si tratta di collegare questi buffer in memoria con i nomi delle risorse da richiamare tramite browser. I nomi delle risorse sono la parte di path locale della URL (o URI), ovvero, in sostanza, il nome del file in una URL completa (Es.: **http://www.mio.it/miapag.html**). In questo ambito minimale, le risorse da richiamare saranno identificate dal solo nome senza estensione. Per cui si tratta di associare una pagina memorizzata con il suo nome Internet corrispondente (per esempio “/indice”).
In realtà questa pagina non si spedirà da sola, e quindi, bisogna collegare il nome della risorsa con una funzione che si occuperà di spedirla. Per rendere il meccanismo più automatico possibile, è stata predisposta una struttura o più esattamente un tipedef chiamato WEBRES. Questa struttura è formata dall'accoppiata di due campi: il nome della risorsa ed il nome della funzione (che in c corrisponde ad un indirizzo). Si tratta, quindi, di formare tante coppie nome-funzione da passare alla funzione getRequest(), che si occuperà di lanciare la funzione corretta (call-back function), o spedire un messaggio standard “Not Found” di risposta, nel caso il nome non combaci con nessuno di quelli predisposti. Di seguito è presentato l'esempio della costruzione di un array di 8 strutture WEBRES ed il suo inserimento nella chiamata alla funzione getRequest():

```
WEBRES rs[8]=
{
{"/index",pindex},
{"/Analog",panalog},
{"/RDigital",rdigital},
{"/Wdigital",wdigital},
{"/wdig",wdig},
{"/Pwm",pwmpage},
{"/PwmSet",pwmset},
{"/End",sessend}
};

void loop()
{
WIFI.getRequest(csocket,8,rs);
:
```

Ogni secondo campo delle strutture corrisponde alla call-back function che getRequest() lancerà. La call-back function si dovrà preoccupare di spedire il buffer corrispondente alla pagina scelta ed il suo prototipo prevede che sia di tipo void (non ritorni nulla) ed abbia un solo argomento: un puntatore ad una null-terminated string fornita dal chiamante (si veda più avanti).

```
void pindex(char *query)
{
WIFI.sendResponse(csocket,pageindex);
}
```

Riassumendo: ponendo getRequest() nel loop, essa si occuperà di tutta la gestione della richiesta. Rileverà la modalità utilizzata GET o POST, comportandosi di conseguenza (i dati sono contenuti in modo diverso) e lancerà la funzione corrispondente alla richiesta (o un messaggio “Not Found”).
La call-back function pindex() sopra descritta però non fa altro che spedire in risposta una pagina Html statica, cioè definita in modo fisso. Arduino Web Server definito così non è molto utile, perché si presuppone di poterlo utilizzare per leggere valori di sensori o attivare uscite. Per ottenere ciò la pagina html di risposta deve essere costruita sul momento contenendo i valori che si vogliono leggere. Cioè una pagina dinamica. Ma sarebbe troppo dispendioso costruire, all'interno delle call-back function, la pagina tutta intera. Per semplificare il compito si è previsto di definire la pagina “una tantum” come una pagina statica, potendo però inserire al suo interno dei tag (etichette segna-posto) nella posizione che si vuole completare al momento. Per questo scopo esiste in alternativa a sendResponse(), la funzione sendDynResponse() che si preoccupa di rintracciare e sostituire i tag mentre spedisce la pagina. La sostituzione avviene sequenzialmente andando a scorrere un array di stringhe predisposto sul momento. Per cui: il primo tag incontrato viene sostituito con la prima stringa dell'array e così via. Il tag utilizzato è il carattere '@'. Ne va usato uno solo a prescindere dalla lunghezza della stringa che lo dovrà sostituire.
Nell'esempio che segue, 3 tag saranno sostituiti da 3 stringhe che rappresentano i valori di 3 input digitali:

```
prog_char pagerdigital[] PROGMEM=
:
"<tr>"
"<td><div align='center'>@</div></td>"
"<td><div align='center'>@</div></td>"
"<td><div align='center'>@</div></td></tr>"
:

void rdigital(char *query)
{
char *val[3];
if(digitalRead(4)) val[0]=ON;else val[0]=OFF;
if(digitalRead(5)) val[1]=ON;else val[1]=OFF;
if(digitalRead(12))val[2]=ON;else val[2]=OFF;
WIFI.sendDynResponse(csocket,pagerdigital,3,val);
}
```

Alla funzione sendDynResponse() va passato l'array delle stringhe e la sua dimensione.
Per far sì che Arduino agisca in seguito a comandi lanciati dal browser (per esempio mediante pulsanti di form), bisogna leggere i dati inviati dalla Request insieme al nome della risorsa. La situazione è diversa se la richiesta è arrivata sotto forma di GET piuttosto che sotto forma di POST (i due metodi cardine del protocollo http). Si tratta, in ogni caso, di utilizzare finalmente quell'argomento passato alla call-back function da getRequest().
Nel primo caso, i dati sono rappresentati da coppie nome-valore che identificano un parametro. I parametri sono accodati al nome della risorsa in un formato che codifica spazi e caratteri speciali e possiamo chiamare query-string. La query-string è fornita sempre (anche se di lunghezza zero) alla call-back function (infatti fa parte del suo prototipo). Allora possiamo utilizzare la funzione getParameter() per recuperare il valore (sempre sotto forma di stringa) del parametro con un certo nome.

```
void pwmset(char *query)
{
char *pwmval;
pwmval=WIFI.getParameter(query,strlen(query),"PWM10");
if (pwmval!=NULL)
{int pv;sscanf(pwmval,"%d",&pv);
analogWrite(10,pv);d10=pv;}
pwmval=WIFI.getParameter(query,strlen(query),"PWM11");
if (pwmval!=NULL)
{int pv;sscanf(pwmval,"%d",&pv);
analogWrite(11,pv);d11=pv;}
pwmpage(query);
}
```

Nel secondo caso, invece, la query-string conterrà valori che possono essere in formato query-string (come in genere fanno le form) oppure in un formato qualunque. Bisogna, comunque, tener presente che il buffer che contiene la query-string è fornito dalla libreria ed ha lunghezza di 64 caratteri (ma ridefinibile con define su HTTPlib.h). I dati in eccesso vengono perduti.
Negli esempi è contenuto un completo Web Server che permette di leggere valori analogici e valori digitali, di attivare e disattivare output digitali e infine di regolare due uscite pwm. Lo sketch è particolarmente compatto (la metà è costituita dalle pagine html in PROGMEM) grazie all'automazione prodotta dalle funzioni getRequest() e sendDynResponse().
Fig. 3, Fig. 4, Fig. 5 (da posizionare a piacere e rimpicciolite in questa zona)
Se, invece, Arduino vuole essere usato come client di un Web Application Server (o di un più semplice CGI), utilizzeremo le funzioni sendRequest() e getResponse(). SendRequest è in effetti costituita da due separate funzioni in base alla modalità che si vuole utilizzare : GET o POST.
Nel caso utilizzassimo sendRequestGET() forniremo sia il nome della risorsa sia i parametri in un'unica query-string. Nel caso utilizzassimo sendRequestPOST() forniremo separatamente il nome della risorsa e i dati collocati in un buffer di tipo null-terminated string.

```
WIFI.addParameter(query,128,"/TestClient",NULL);
WIFI.addParameter(query,128,"A1",sa1);
:
WIFI.sendRequestGET(csocket,query);
```

oppure

```
sprintf(rec,"%d %d %d %d",an1,an2,d1,d2);
WIFI.sendRequestPOST(csocket,"/TestClient",rec);
```

Nel caso di utilizzo di sendRequestGET(), la query-string è stata formata con l'aiuto della funzione addParameter(). La prima volta inizializzando la query-string con il nome della risorsa (valore null), e poi con le coppie nome-valore per i singoli parametri.
La funzione getResponse(), è quella da utilizzare per recuperare la risposta dal server. Questa può essere anche formata da numerosi dati in vari formati : dalla pagina HTML, a dati in formato XML, JSON o csv (comma separated values).
Nel caso i dati non possano essere contenuti in un unico buffer, è possibile richiamare la funzione getNextResponseBuffer() in un loop fino a che non ritorni 0.

Modulo transceiver Wi-Fi a 2,4 GHz standard 802.11 IEEE della Microchip. Dispone di antenna integrata su PCB e supporto integrato per l'hardware AES, TKIP e (WEP, WPA , WPA2). Progettato per essere utilizzato con lo stack TCP / IP della Microchip . Il software dello stack ha un driver integrato che implementa le API e che viene utilizzato nei moduli di comando e controllo, e per la gestione del traffico dati a pacchetto. Il modulo MRF24WB0MA può essere interfacciato a centinaia di microcontrollori PIC® tramite interfaccia SPI a 4 fili ed è una soluzione ideale per dispositivi a basso consumo, reti di sensori Wi-Fi a basso data-rate, domotica e applicazioni consumer.

Documentazione e link utili

- ***Libreria***