



Università Politecnica delle Marche
Ingegneria Informatica e dell'Automazione
Anno Accademico 2019-20120

BallBot

Comunicazione Wireless

Gruppo 7

Danilo Gervasio

danielodangrad@gmail.com

Piero Castriota

castrio98@gmail.com

Luciano Frick

lucio.frick@gmail.com

Lorenzo Deplano

deplano.lorenzo97@gmail.com

Tommaso Coricelli

coricellitommaso@gmail.com

Franci Pelivani

s1078913@studenti.univpm.it

Relatore:

Prof. Andrea Bonci

Indice

1. INTRODUZIONE AL PROGETTO.....	4
1.1 Cos'È UN BALLBOT?	5
1.2 IL NOSTRO CONTRIBUTO AL BALLBOT - LA COMUNICAZIONE Wi-Fi.....	6
2. STRUMENTAZIONE	9
2.1 RENESAS DEMONSTRATION KIT YRDKRX63N v3.2	9
2.2 MODULO WiFi ESPRESSIF ESP32 WROOM.....	9
2.3 OSCILLOSCOPIO	9
2.4 PC CON CAVO USB TO SERIAL.....	10
3. FASI PROGETTUALI	11
3.1 SCELTA DEL CANALE SCIC	11
3.2 IMPOSTAZIONE DEI REGISTRI	13
3.3 CREAZIONE DEGLI INTERRUPT DI TRASMISSIONE E RICEZIONE	16
3.4 TRASMISSIONE SCHEDA-OSCILLOSCOPIO	17
3.5 TRASMISSIONE E RICEZIONE TRAMITE COM	18
3.6 CONFIGURAZIONE DEL MODULO WiFi	21
3.6.1 <i>Schematics modulo ESP-WROOM-32</i>	21
3.6.2 <i>Funzionamento</i>	21
3.7 COMPOSIZIONE DEL PACCHETTO E COMANDI BALLBOT	23
3.8 TRASMISSIONI E RICEZIONE CON MODULO WiFi	25
3.8.1 <i>Impostazione del modulo</i>	25
3.8.2 <i>Elaborazione iniziale dei dati ricevuti dal modulo</i>	26
3.8.3 <i>Controllo dell'header ricevuto.....</i>	27
3.8.4 <i>Controllo dell'header per risposte dei comandi AT.....</i>	28
3.8.5 <i>Parsing del comando</i>	29
4. APPENDICE.....	30
PROJECTLABORATORIO.C.....	30
DATA_PARSER.H.....	30
BALLBORT_UART.H.....	31
BALLBOT_COMMAND.H	32
WIFI_SETTINGS.H.....	32
BALLBOT_UART.C.....	33
DATA_PARSER.C	42
5. CONCLUSIONI	46
6. BIBLIOGRAFIA.....	47

1. Introduzione al progetto

*Ciò che non puoi comunicare
rovina la tua vita.*

(Robert Anthony)

Questa relazione vuole porsi come guida step-by-step da seguire per generare e comprendere una semplice comunicazione Wi-Fi utile alla trasmissione di comandi da un controller remoto (computer, tablet o cellulare) ad una scheda *Renesas RX63N*.

Nel seguito verranno prima descritte brevemente le strumentazioni usate e successivamente saranno ripercorsi tutti i passi che hanno portato alla realizzazione finale del progetto.

Lo scopo della relazione (oltre ai fini di esame) è quello di guidare i futuri lettori alla realizzazione di un programma analogo in tempi minori.

1.1 Cos'è un BallBot?

Negli ultimi anni il settore della robotica ha subito una notevole espansione, specialmente a causa dell'innovazione sempre più significativa dei sensori e dei sistemi controllo.

Vediamo sempre più comunemente robot capaci di volare, robot acquatici, robot antropomorfi... Un particolare successo hanno avuto i robot capaci di rimanere in equilibrio su due ruote e muoversi su di esse.

Il **BallBot** nasce dall'idea di ridurre i punti di appoggio del robot ad uno solo.

La sua stabilità dinamica lavora sugli stessi principi del pendolo inverso, che unito al fatto di avere un unico punto di contatto con il suolo lo rendono un robot molto agile e facile da manovrare in luoghi stretti o affollati.

A differenza di altri robot a due ruote, come il *Segway* ad esempio, il BallBot non ha un raggio di sterzata e non ha bisogno di curvare per cambiare direzione, rendendolo capace di lavorare su di un piano omnidirezionale.

Il robot è composto da due parti principali:

- il corpo (o body), su cui sono installate tutte le parti meccaniche ed elettroniche del robot.
- la sfera, l'unico punto di contatto con il suolo.

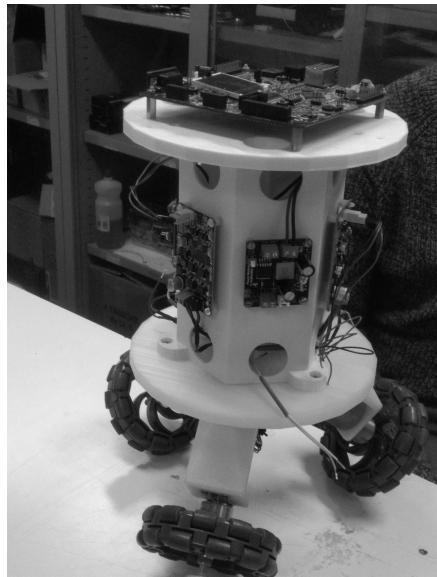
Il corpo del BallBot non è legato in alcun modo alla sfera. Esso è libero di muoversi ed oscillare.

Il contatto tra le due parti avviene attraverso tre ruote, che ruotando ne garantiscono il movimento.

L'utilizzo di normali ruote gommate non avrebbe permesso la rotazione della palla in ogni direzione, per questo si utilizza un particolare tipo di ruote: le ruote omnidirezionali.

Questa tipologia di ruote è formata da tanti piccoli dischi, disposti lungo la circonferenza principale e orientati perpendicolarmente alla direzione di rotazione. L'effetto che si ottiene è che la ruota oltre che sviluppare una forza lungo la direzione di movimento, può anche scorrere lateralmente se viene applicata qualche forza longitudinale.

L'aspetto controintuitivo del BallBot è che per muoversi in avanti, il corpo deve andare in avanti, e per muovere il corpo in avanti la palla deve ruotare all'indietro. Inoltre, il ballbot deve inclinarsi per compensare le forze centripete, che si traducono in movimenti eleganti e aggraziati.



1.2 Il nostro contributo al BallBot - la comunicazione Wi-Fi

Lo scopo del progetto è quello di comunicare con la scheda Renesas RX63N montata sul BallBot da remoto fornendo dei comandi base.

La comunicazione avviene tramite Wi-Fi con lo **standard RS232** passando attraverso il modulo **ESP-WROOM-32**.

Questo tipo di comunicazione viene detta “*seriale*” poiché permette la trasmissione e la ricezione (anche simultanea) dei dati un bit alla volta, sequenzialmente.

Il supporto fisico richiesto dunque è di complessità minima (pochi conduttori, di solito 3 o 4).

Nei sistemi di trasmissione seriale, a livello di collegamento tra periferiche, si distinguono due tipologie di trasmissione:

- **Trasmissione asincrona**
- **Trasmissione sincrona**

La comunicazione usata in questo progetto è di tipo asincrono, ormai diventata uno standard per le trasmissioni da punto a punto a bassa velocità, poiché richiede un supporto fisico molto semplice (al massimo due fili) e anche se la velocità di comunicazione risulta limitata rispetto a quello sincrona è comunque più che sufficiente.

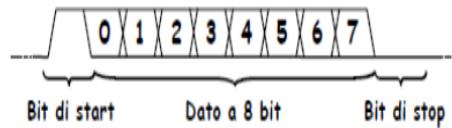
Il termine “*asincrono*” indica che la comunicazione avviene senza sincronismo dei clock del *trasmettitore (Tx)* e del *ricevitore (Rx)*. Il ricevitore deve perciò estrarre il sincronismo dalla stessa linea riservata al trasferimento dei dati.

Perché ciò accada serve che i pacchetti scambiati rispettino dei protocolli di formato, con alcuni bit necessari a far capire l'inizio e la fine della trasmissione, chiamati appunto “**bit di start**” e “**bit di stop**”.

La comunicazione seriale asincrona è ottima nel caso di trasferimento di informazioni che sopraggiungono in modo “sporadico”, con pause anche lunghe tra un dato e l'altro.

In generale i pacchetti che vengono scambiati rispettano questa struttura standard:

- *Bit di start*: 1, sempre segnalato da una transizione dallo stato inattivo a quello opposto
- *Bit dei dati*: da 5 a 8
- *Bit di parità*: nessuno o 1
- *Bit di stop*: 1, 1.5, 2



L'impostazione di questi parametri normalmente avviene con l'ausilio di apposite periferiche denominate **UART**, acronimo di *Universal Asynchronous Receiver/Transmitter*.

Il bit di parità non è obbligatorio. Può essere trasmesso subito dopo ciascun dato per introdurre un semplice sistema di controllo sugli eventuali errori di ricezione. Se la somma dei bit trasmessi è pari, il bit è posto a 1, altrimenti a 0 (o viceversa).

Il ricevitore calcola il bit di parità sui bit ricevuti (ad esempio fa una somma dei bit, o fa altre operazioni su essi). Se questo risulta uguale a quello trasmesso considera la ricezione esente da errore (nel caso di due errori la comunicazione verrebbe considerata esente da errori, ma questa situazione è assai meno probabile) altrimenti segnala la situazione di errore.

L'utilizzo di bit trasmessi per la sincronizzazione e la correzione di eventuali errori comporta l'abbassamento della velocità effettiva di trasferimento dei dati.

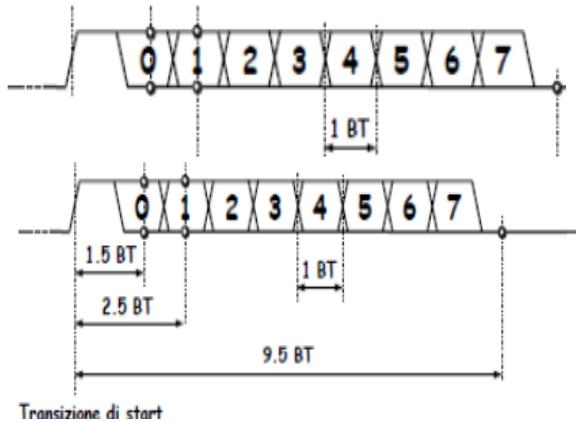
Se ad esempio si vogliono trasferire 8 bit saranno necessari in realtà ben 10 bit, che comporta un sotto-utilizzo del canale del 20%.

Questo è un limite strutturale della trasmissione seriale asincrona.

L'intervallo di bit ha durata **BT** pari a 1/bps (il bps è detto anche **baud rate**) dove il bps è la velocità di trasmissione impostata.

Affinché la trasmissione abbia successo, è necessario che il ricevitore campioni i bit dei dati a metà dell'intervallo di bit.

Nel caso di dati a 8 bit, gli istanti di campionamento ideali per il ricevitore sono quindi dopo 1.5 BT, 2.5 BT, C, 8.5 BT dalla transizione di start (a 9.5 BT avverrà la ricezione dello STOP). Bisogna ricordare però che i due clock (Tx e Rx) non sono sincronizzati.



Se il sistema funziona correttamente, dopo un certo numero di BT (che dipende dalla quantità di bit scambiati) dalla transizione di start il ricevitore si deve trovare la linea ad un livello corrispondente al bit di stop. Altrimenti si può dedurre che si è verificato un errore di sincronismo, che dovrà essere segnalato.

La scheda Renesas utilizzata è la RX63N; questa ha 13 interfacce seriali indipendenti (**SCI**, *Serial Communication Interface*).

La **SCI** è configurata in due tipologie di moduli:

- **SCIc** (da *SCI0* a *SCI11*) : possono gestire comunicazioni seriali asincrone e comunicazioni sincrone con clock.
- **SCIId** (*SCI12*) : oltre che gestire le comunicazioni seriali asincrone, supporta un protocollo seriale con una struttura formata da uno Start Frames e Information Frames.

La comunicazione asincrona può essere gestita mediante la periferica **UART** (*Universal Asynchronous Receiver/Transmitter*: mediante un solo conduttore utilizzato sia per trasmettere che per ricevere) o mediante **ACIA** (*Asynchronous Communications Interface Adapter*). In tale progetto si è utilizzata la periferica **UART**.

SCI supporta anche smart card (*IC card*), *I2C*, *SPI interface*.

2. Strumentazione

Per la realizzazione della connessione Wi-Fi del Ballbot è stato necessario l'utilizzo di una strumentazione varia, tra componenti definitivi e componenti utilizzati a fine di testare il funzionamento del sistema.

2.1 Renesas Demonstration Kit

YRDKRX63N v3.2

E' la scheda utilizzata durante il corso, su cui è montato il microcontrollore **RX63N** che permette di testare la suite **Renesas** per la quale è possibile la programmazione, la codifica e il debugging mediante l'*IDE e2studio*.



2.2 Modulo WiFi Espressif ESP32

WROOM

Questo componente permette la comunicazione wireless verso l'ambiente, viene collegato alla scheda Renesas tramite i pin di trasmissione e ricezione. Configurando tramite il firmware proprietario la scheda Wi-Fi è possibile connettere uno o più dispositivi abilitandoli ad inviare semplici **comandi AT** al microcontrollore.



2.3 Oscilloscopio

Utilizzato per testare i segnali di test inviati dal microcontroller, l'oscilloscopio è stato lo strumento iniziale per comprendere il funzionamento della comunicazione seriale.



2.4 PC con cavo USB to SERIAL

L'ausilio di un computer con inserito un cavo adattatore da USB a SERIAL è servito per il test di funzionamento dei canali di ricezione e trasmissione programmati nel microcontrollore, il collegamento è stato effettuato tramite un adattatore di tensione da 5V a 3.3V collegato ai pin della scheda al fine di mantenere la tensione di utilizzo della Renesas.

3. Fasi progettuali

3.1 Scelta del canale SCIc

Inizialmente si è dovuto scegliere uno dei 12 canali ($SCIc$, $c=0,..,12$) che la scheda Renesas offre per l'implementazione della comunicazione seriale. La decisione è stata effettuata consultando le seguenti tabella dell'hardware manual (pag 1319).

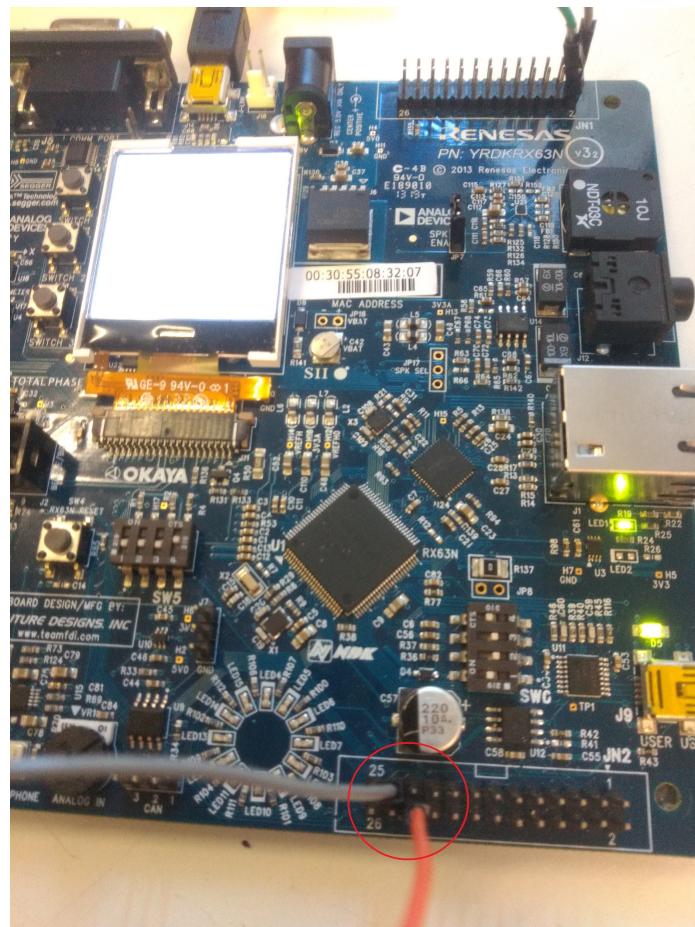
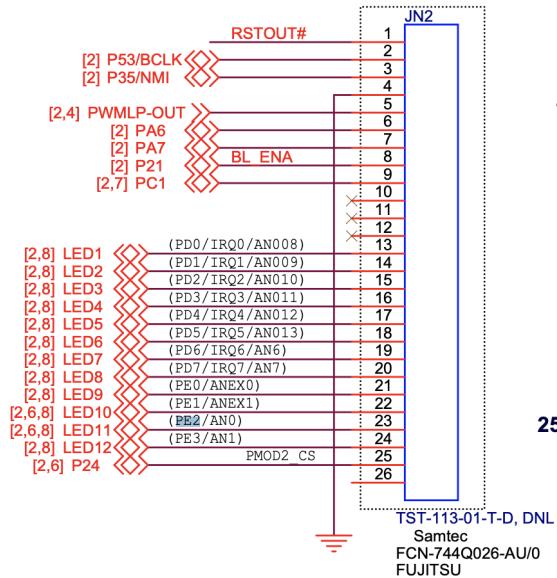
Table 35.3 List of Functions of SCI Channels

Item	SCI0 to SCI4, SCI7 to SCI11	SCI5, SCI6	SCI12
Asynchronous mode	○	○	○
Clock synchronous mode	○	○	○
Smart card interface mode	○	○	○
Simple I ² C mode	○	○	○
Simple SPI bus	○	○	○
Extended serial mode	—	—	○
TMR clock input	—	○	○

Tale tabella ci assicura che il canale $SCI12$ è utilizzabile per la comunicazione asincrona. Inoltre si è dovuto verificare se i canali TXD e RXD corrispondenti al canale $SCI12$ ($TXD12$ e $RXD12$) siano collegati a dei pin utilizzabili sulla scheda Renesas (ad esempio sui connettori $JN1$ o $JN2$). La tabella successiva mostra che questi due canali corrispondono rispettivamente alla porta $PE1$ e $PE2$ della scheda.

SCI12	RXD12 (input)/ SMISO12 (input/output)/ SSCL12 (input/output)/ RXDX12 (input)	PE2	○	○	○	○	○	○
	TXD12 (output)/ SMOSI12 (input/output)/ SSDA12 (input/output)/ TXDX12 (output)/ SIOX12 (input/output)	PE1	○	○	○	○	○	○

Si è utilizzato lo schematics della scheda per capire a quali pin corrispondessero queste porte (porta E bit 1 e 2). Il canale di trasmissione $TXD12$ si trova sul pin 22 del connettore $JN2$, il canale di trasmissione $RXD12$ si trova sul pin 23 di tale connettore.



Pin 22 e 23 del connettore JN2 della scheda

3.2 Impostazione dei registri

La comunicazione seriale è stata gestita mediante la libreria *ballbot_uart.c*. Tale libreria implementa una funzione di inizializzazione (*uart_init()*) che imposta i registri della scheda Renesas per una comunicazione seriale asincrona a 8 bit, senza bit di parità ed un solo bit di stop; richiamata dal main del programma).

Il canale scelto per la comunicazione implementa i seguenti registri utilizzati per la trasmissione delle informazioni:

- **TDR (Transmit Data Register)** : registro di 8 bit che memorizza singolarmente i byte da trasmettere. Quando il registro *TSR* è vuoto si genera un interrupt di trasmissione *TXI* (se abilitato in precedenza sul rispettivo canale utilizzato). In tal caso viene spostato il byte dal registro *TDR* al registro *TSR* del canale *SCIc* utilizzato. Il dato è successivamente inviato dal registro *TSR* al *TXDn* pin (nel nostro caso *TXD12* corrispondente al pin 22 del connettore *JN2*). La CPU può scrivere o leggere dati dal *TDR* in qualsiasi momento.
 - **TSR (Transmit Shift Register)** : come detto in precedenza, questo è il registro nel quale transitano i dati prima di essere trasmessi sul rispettivo *TXDn* pin della scheda. Lo stato di tale registro è responsabile della generazione dell'interrupt di trasmissione *TXI*.
 - **RDR (Receive Data Register)** : registro di 8 bit che memorizza i dati in ricezione. Quando il canale *SCIc* riceve un frame di dati, questi sono trasferiti dal registro *RSR* al registro *RDR*. I dati sono letti nel registro *RDR* solamente quando si genera l'interrupt *RXI*, ovvero quando si ricevono dei dati nel registro *RSR*.
- Tale registro può essere sovrascritto dalla CPU.
- **RSR (Receive Shift Register)**: registro utilizzato per ricevere i dati seriali dal *RXDn* pin (nel nostro caso *RXD12* corrisponde al pin 23 del *JN2*). Non appena il registro riceve dei dati questo genera un interrupt *RXI* e li sposta sul registro *RDR*.

Sono stati inizializzati i seguenti registri:

- **SCR (Serial Control Register)** : contiene i bit per il controllo della comunicazione.

b7	b6	b5	b4	b3	b2	b1	b0
TIE	RIE	TE	RE	MPIE	TEIE	CKE[1:0]	

st: 0 0 0 0 0 0 0 0

- TIE (Transmit Interrupt Enabled):** abilita o disabilita la richiesta dell'interrupt di trasmissione *TXI*. Impostato a 1 nel codice

- RIE** (*Receive Interrupt Enabled*): abilita o disabilita la richiesta dell'interrupt di ricezione *RXI*. Impostato a 1 nel codice
- TE** (*Transmit Enabled*): abilita o disabilita la trasmissione sul canale seriale scelto. Impostato a 1 nel codice
- RE** (*Receive Enabled*): abilita o disabilita la ricezione sul canale seriale scelto. Impostato a 1 nel codice.
- TEIE** (*Transmit End Interrupt Enabled*): abilita o disabilita la richiesta dell'interrupt di fine trasmissione *TEI*. Impostato a 0 nel codice.
- CKE** (*Clock Enabled*): abilita o disabilita l'utilizzo di un clock sorgente. Impostato a 0 nel codice in quanto si sta utilizzando una comunicazione asincrona.
- **SMR** (*Serial Mode Register*) : contiene i bit per l'impostazione delle modalità nel quale deve avvenire la comunicazione. Tutti i bit di tale registro sono stati impostati a 0 nel codice.

	b7	b6	b5	b4	b3	b2	b1	b0
t:	0	0	0	0	0	0	0	0

- CKS** (*Clock Selects*): bit che selezionano la sorgente di clock
- MP** (*Multi Processor Mode*): abilita/disabilita la modalità multi-processor; in tale modalità i bit *PE* e *PM* sono disabilitati.
- STOP** (*Stop Bit Length*): imposta la quantità di stop bit nella fase di trasmissione; per la ricezione viene considerato solamente uno stop bit.
- PM** (*Parity Mode*): seleziona la parità di controllo per la trasmissione e la ricezione
- PE** (*Parity Enable*): se il bit di tale registro è 1 si aggiunge il bit di parità ai dati da trasmettere e si controlla la parità ai dati ricevuti.
- CHR** (*Character Length*): seleziona la lunghezza dei dati da trasmettere e ricevere. Se assume valore 0 si utilizza un multiplo di un byte come lunghezza dati.
- CM** (*Communications Mode*): seleziona la modalità di comunicazione, 0 equivale ad una comunicazione asincrona mentre 1 corrisponde ad una comunicazione clock sincrona.
- Si è impostato il bit **SMIF** (*Smart Card Interface Mode Select*) del registro **SCMR** (*Smart Card Mode Register*) al valore 0, indicante una modalità di interfaccia per una comunicazione seriale.

- **BRR** (*Bit Rate Register*): registro a 8 bit che imposta il bit rate della comunicazione. Ogni canale *SCI* ha un controller del baud rate indipendente, infatti questi canali possono essere impostati con bit rates differenti fra loro. Il valore N da assegnare a questo registro dipende dal **PCLK** (*Peripheral Clock Frequency*, la massima frequenza al quale lavora la periferica Renesas, solitamente 48 MHz) e dalle modalità di della scheda (pag. 1339 hardware manual).

Table 35.8 Relationships between N Setting in BRR and Bit Rate B

Mode	ABCS Bit in SEMR	BRR Setting	Error
Asynchronous, multi-processor transfer	0	$N = \frac{PCLK \times 10^6}{64 \times 2^{2n-1} \times B} - 1$	$\text{Error (\%)} = \left\{ \frac{PCLK \times 10^6}{B \times 64 \times 2^{2n-1} \times (N+1)} - 1 \right\} \times 100$
	1	$N = \frac{PCLK \times 10^6}{32 \times 2^{2n-1} \times B} - 1$	$\text{Error (\%)} = \left\{ \frac{PCLK \times 10^6}{B \times 32 \times 2^{2n-1} \times (N+1)} - 1 \right\} \times 100$
Clock synchronous, simple SPI		$N = \frac{PCLK \times 10^6}{8 \times 2^{2n-1} \times B} - 1$	
Smart card interface		$N = \frac{PCLK \times 10^6}{S \times 2^{2n+1} \times B} - 1$	$\text{Error (\%)} = \left\{ \frac{PCLK \times 10^6}{B \times S \times 2^{2n+1} \times (N+1)} - 1 \right\} \times 100$
Simple I ² C ⁺¹		$N = \frac{PCLK \times 10^6}{64 \times 2^{2n-1} \times B} - 1$	

B: Bit rate (bps)

N: BRR setting for baud rate generator (0 ≤ N ≤ 255)

PCLK: Operating frequency (MHz)

n and S: Determined by the SMR setting listed in the following table.

Note 1. Adjust the bit rate so that the widths at high and low level of the SCL output in simple I²C mode satisfy the I²C standard.

Per l'inizializzazione di tale registro si è presa in considerazione la formula descritta nella modalità asincrona con *ABCS* bit nel registro *SEMR* pari a 0 (il registro *SEMR* non è stato inizializzato quindi si assume che tutti i suoi bit siano nulli).

Il valore n è determinato dalle impostazioni date al registro *SMR*, citato precedentemente. Nel nostro caso n assume valore 0 e si è scelto un bit rate pari a 115200 bit/s, concorde con il baud rate del modulo WiFi utilizzato. La formula è la seguente:

```
SCI12.BRR=48000000 / ((64/2) * 115200) -1;
```

All'interno della funzione *uart_init()* sono stati inizialmente disabilitati gli interrupt **TX** (*Transmission interrupt*) , **RX** (*Reception interrupt*) e **TEI** (*Transmission end interrupt*), disabilitandoli nella periferica *SClC* scelta attraverso la macro *IR(periferica, interrupt)* e nell'**ICU** (*Interrupt Control Unit*) attraverso la macro *IEN(periferica, interrupt)*.

```

/* Clear IR bits (of the interrupt) for SCI12 registers: TXI, RXI, TEI */
IR(SCI12,RXI12)=0; //Clear any pending ISR
IR(SCI12,TXI12)=0;
IR(SCI12,TEI12)=0;

IPR(SCI12,RXI12)=4; /* Set interrupt priority, not using the transmit end interrupt */
IPR(SCI12,TXI12)=4;

/* Disable all the interrupt (TXI,RXI,TEI) */
IEN(SCI12,TXI12)=0;
IEN(SCI12,RXI12)=0;
IEN(SCI12,TEI12)=0;

```

3.3 Creazione degli interrupt di trasmissione e ricezione

Per gestire i dati trasmessi e ricevuti dalla scheda si è creata una struttura *queue*, definita nel file *ballbot_uart.h*, contenente un array di caratteri dove sono memorizzati i dati trasmessi/ricevuti, un puntatore al primo elemento dell'array (*head*), un puntatore all'ultimo elemento dell'array (*tail*) e il numero di elementi presenti nel vettore. La sua dimensione è impostata attraverso la *define DIM_BUFFER* all'interno del file *wifi_settings.h*.

```

/* struct used for managing the data of the communication */
typedef struct {
    unsigned char data [DIM_BUFFER]; /* chars buffer used with a circular nature */
    unsigned int head; /* pointer to the first item of the buffer */
    unsigned int tail; /* pointer to the last item of the buffer */
    unsigned int counterElements;
}queue;

```

L'interrupt di trasmissione posiziona nel registro *SCI12.TDR* la testa del vettore dati contenuto nella coda di trasmissione, quindi incrementa il puntatore della testa e decrementa il numero di elementi presenti nella coda di trasmissione. Se l'interrupt di trasmissione è abilitato, questo sarà eseguito nuovamente fino a che tutti gli elementi della coda di trasmissione non sono stati trasmessi (*tx_queue_pointer->counterElements = 0*)

```

/* Function name: SCI8_TXI8_int */
#pragma interrupt SCI12_TXI12_int (vect = VECT_SCI12_TXI12, enable)
static void SCI12_TXI12_int()
{
    if (!queue_is_empty(tx_queue_pointer)) /* if the tx queue is not empty */
    {
        SCI12.TDR=tx_queue_pointer->data[tx_queue_pointer->head]; /* putting into the data buffer
register (TDR) of the SCI8 channel the first element of the data buffer trasmission */

        tx_queue_pointer->head++; /* Now the head of the buffer will be the next element */
        tx_queue_pointer->counterElements--;
        if (tx_queue_pointer->counterElements==0) /* Disable the tx interrupt if all the elements have been transmitted */
        {
            interrupt_tx_disable();
            queue_clear(tx_queue_pointer); /* Clear the tx queue after the transmission */
        }
    }
}

```

L'interrupt di ricezione preleva il byte presente nel registro *SCI12.RDR* e lo posiziona nell'ultima posizione della coda di ricezione, successivamente si incrementano il numero di elementi della coda. Se la coda è piena o l'ultimo elemento della coda è il carattere *line feed* ('\n', carattere 10 decimale) allora significa che la ricezione è terminata e si è pronti per processare i dati ricevuti.

Successivamente si spiegherà perché l'ultimo carattere del comando ricevuto deve terminare con il carattere *line feed*.

```
uint8_t read_byte=SCI12.RDR; /* Read byte from RDR register */
/* Putting the data from RDR register into the reception buffer */
rx_queue_pointer->data[rx_queue_pointer->tail]= read_byte;
rx_queue_pointer->counterElements++; /* Incrementing the counter
rx_queue_pointer->tail++; /* Incrementing the tail pointer to the next position
```

Tale interrupt sarà eseguito fino a che non si sono ricevuti tutti i byte trasmessi dal trasmittente (nel nostro caso il modulo *WiFi* riceverà dei comandi in *WiFi* che trasmetterà attraverso la seriale alla scheda Renesas).

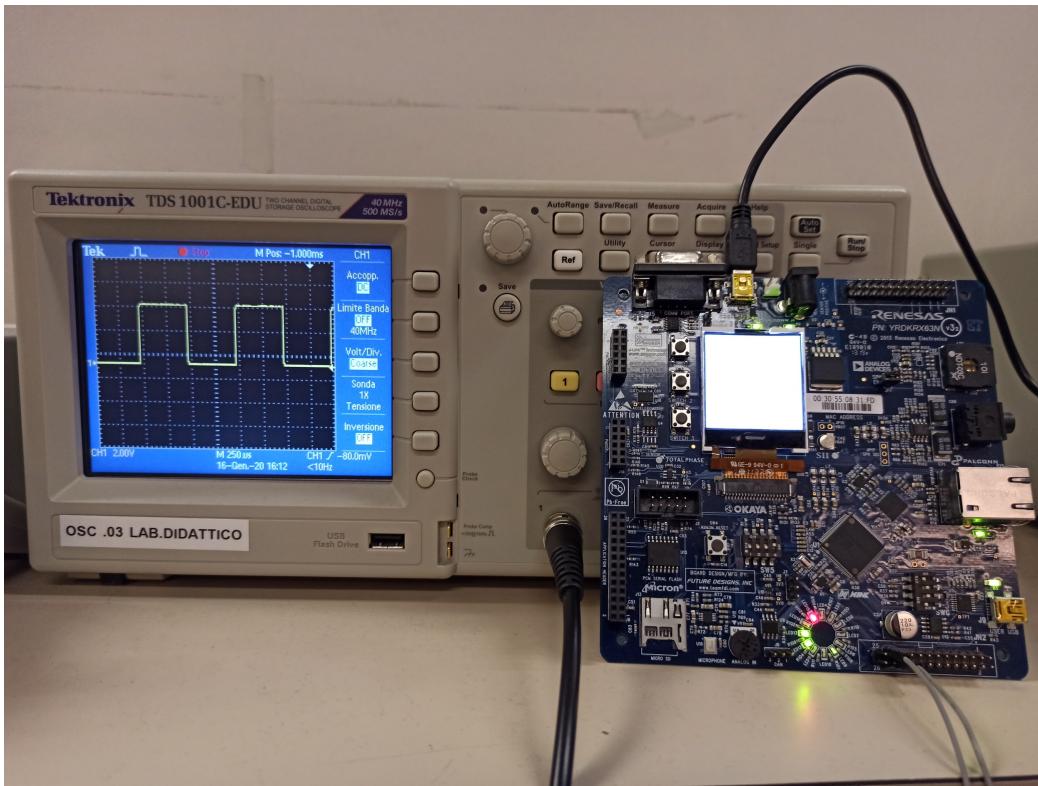
All'interno degli interrupt si utilizzano delle funzioni propedeutiche al totale funzionamento della comunicazione:

- *interrupt_tx_init ()* → imposta i registri per abilitare l'interrupt di trasmissione
- *interrupt_rx_init ()* → imposta i registri per abilitare l'interrupt di ricezione
- *interrupt_tx_disable ()* → imposta i registri per disabilitare l'interrupt di trasmissione
- *interrupt_rx_disable ()* → imposta i registri per disabilitare l'interrupt di ricezione
- *queue_is_full (queue*)* → controlla se la coda passata come parametro è piena
- *queue_is_empty (queue*)* → controlla se la coda passata come parametro è vuota
- *queue_clear (queue*)* → resetta la coda passata come parametro

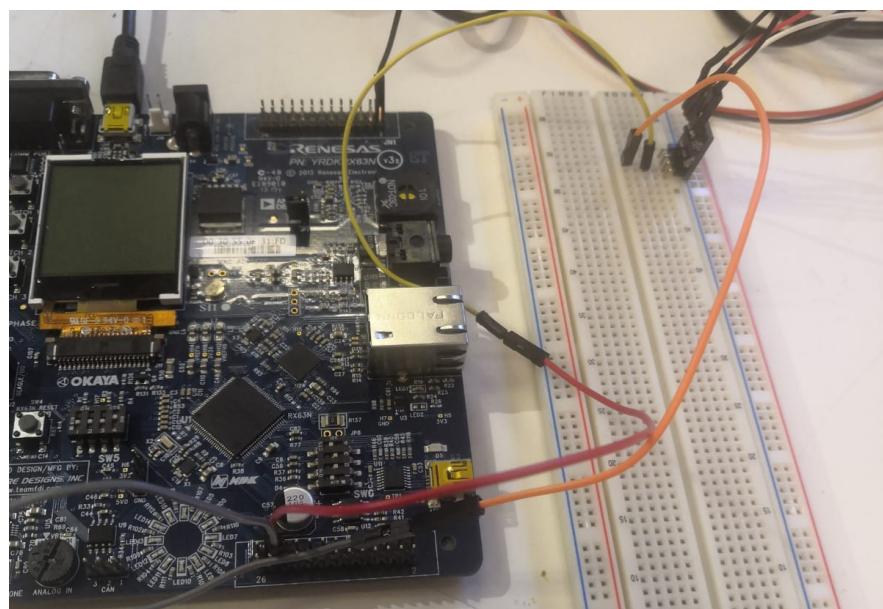
3.4 Trasmissione scheda-oscilloscopio

Il primo passo per impostare la comunicazione è stato quello di trasmettere una serie di byte periodicamente dalla scheda ad un oscilloscopio che aveva il solo

scopo di rilevare un eventuale segnale in entrata. Tale segnale è stato trasmesso dalla scheda Renesas attraverso la *comunicazione seriale*, utilizzando l'*interrupt di trasmissione*.



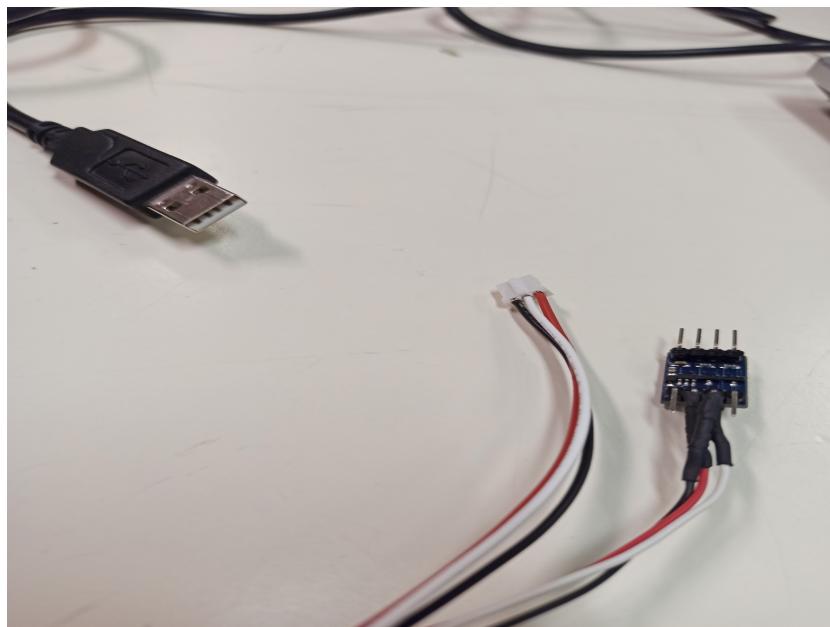
3.5 Trasmissione e ricezione tramite COM



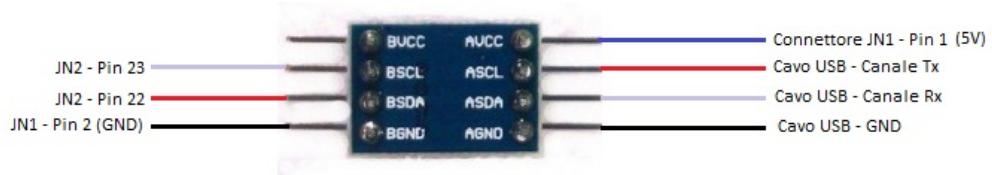
Per progettare e verificare il corretto funzionamento della comunicazione seriale sia in trasmissione che in ricezione dalla scheda Renesas si è scelto di interfacciarsi con un PC tramite il software **Docklight**.

Docklight è un programma di testing per diversi protocolli di comunicazione tra cui quella interfacciata tramite *porta seriale COM* (utilizzata in questo caso). Il tipo di comunicazione che imposteremo su Docklight sarà *RS232 Terminal / RS232 Monitor*.

L’interfaccia fisica consiste in un cavo con un terminale USB e un terminale adatto al collegamento con i connettori sulla scheda.

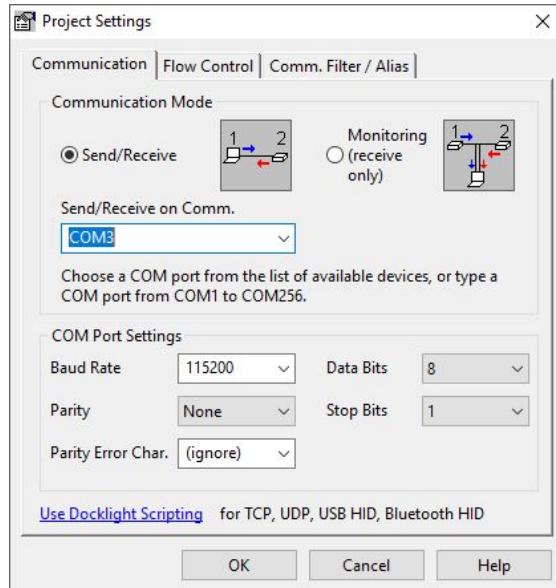


La porta USB del PC trasmette segnali tramite il cavo con livello logico alto di tensione pari a 5V, differente da livello alto di tensione della *scheda Renesas*, pari a 3.3V. Per questo è necessario interporre tra il cavo e il connettore un **traslatore di livello** secondo lo schema seguente:



Il canale A del traslatore è alimentato a 5V e gestisce i segnali a 5V che arrivano dal cavo, l’alimentazione passa al canale B e viene attenuata a 3,3V e così viene fatto anche per i segnali di trasmissione e ricezione.

Una volta completato il collegamento, è necessario configurare Docklight. Tramite la finestra “*Project Settings*” specifichiamo attraverso quale porta USB del computer avverrà la comunicazione (una volta collegato il cavo essa verrà riconosciuta automaticamente e le verrà assegnato un numero) ed impostiamo i parametri nel seguente modo:



Questi parametri sono gli stessi che sono stati impostati sulla scheda.

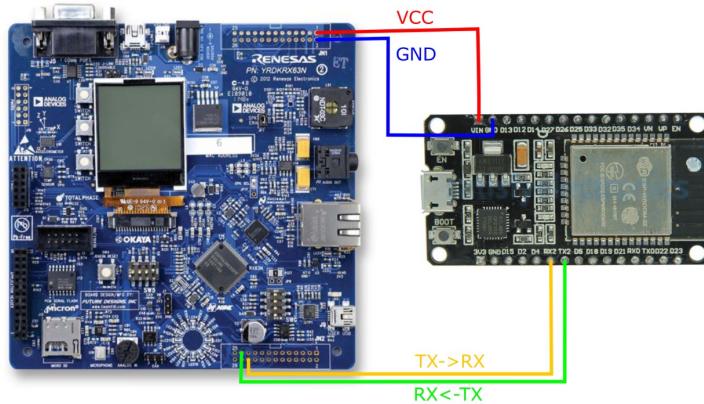
È stata impostata la **Baud Rate** a 115200 perchè è quella di default utilizzata dal modulo Wi-Fi, che verrà trattato in seguito.

Avviando la comunicazione tramite il tasto “*Play*” di Docklight si può cominciare ad inviare e ricevere. Come primi test sono stati inviati e ricevuti singoli caratteri (e quindi singoli byte), per poi arrivare ad inviare e ricevere stringhe più complesse.

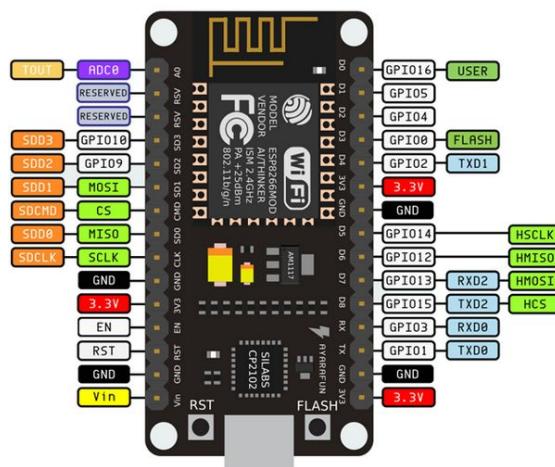
All’invio di ogni carattere da Docklight si attiva l’interrupt di ricezione della scheda e viene eseguito il sottoprogramma corrispondente a tale interrupt (ciò si è verificato utilizzando dei breakpoint interni al sottoprogramma). Viceversa invocando l’interrupt di trasmissione dalla scheda è possibile leggere su Docklight ciò che è stato trasmesso. Tutto ciò è stato usato per studiare e verificare il comportamento della comunicazione seriale configurata sulla scheda e procedere con le fasi successive. Questa fase ci ha permesso quindi di stabilire una semplice comunicazione seriale tra computer e scheda. Lo stesso sarà poi fatto con il modulo Wi-Fi per studiare che tipo di pacchetti sono inviati dalla sua porta seriale una volta ricevuti dal terminale connesso.

3.6 Configurazione del modulo Wi-fi

3.6.1 Schematics modulo ESP-WROOM-32



PIN DEFINITION



3.6.2 Funzionamento

Il modulo Wi-Fi contiene una memoria flash nella quale è installato un firmware che permette di inviargli dei comandi specifici, chiamati **comandi AT (A**Ttention). Questo è uno specifico set di comandi originariamente sviluppato per il modem Hayes Smartmodem. E' necessario configurare il modulo come server TCP per connessioni multiple, questo è possibile utilizzando dei comandi AT specifici.

La stringa di inizializzazione consiste in una serie di comandi che prepara il modulo per la comunicazione, impostando caratteristiche come il tipo di connessione, i tempi di attesa, la rilevazione del segnale di occupato, ecc. Nei modelli di modem più recenti l'inizializzazione viene gestita da programmi con una interfaccia grafica, in modo che l'utente non debba digitare questi comandi manualmente, per questo abbiamo impostato la scheda Renesas in maniera tale che invii i comandi AT al modulo Wi-Fi automaticamente.

Ogni funzione del modulo è governata dal relativo comando AT. Per inviare un comando occorre trasmettere sulla porta seriale del modulo una stringa ASCII formata da AT seguito da uno o più comandi e dal carattere carriage (CR) e line feed (LF); questi due caratteri identificano la terminazione del pacchetto. Ad esempio:

AT+CWMODE=3 <CR><LF>

Durante tutta la fase di programmazione iniziale abbiamo trasmesso i comandi AT tramite il software **Docklight**, mediante l'utilizzo del cavo **COM**.

Quando **ESP32** lavora come TCP server, la connessione multipla è abilitata; quindi più di un client ha la possibilità di connettersi.

I comandi che ci sono serviti sono:

- *AT+CWMODE=3<CR><LF>* → per settare il modulo in **WiFi mode**;
- *AT+CIPMUX=1<CR><LF>* → per abilitare la connessione multipla;
- *AT+CIPSERVER=1<CR><LF>* → per configuralo come **TCP server**.

Per ogni comando inviato il modulo invia 3 risposte, supponiamo ad esempio il comando *AT+CWMODE=3<CR><LF>*:

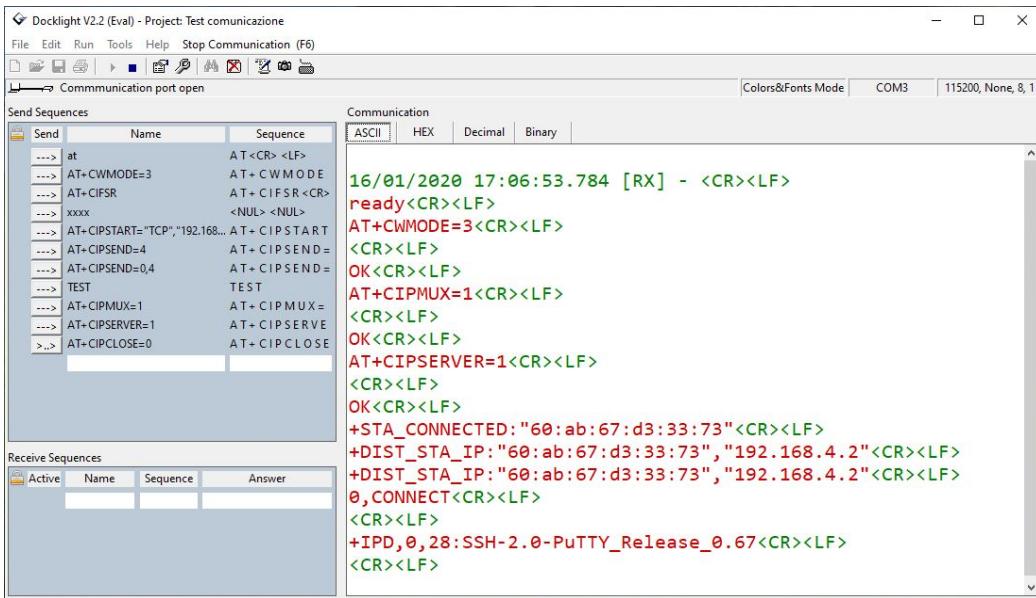
- *AT+CWMODE=3<CR><LF>* → echo del comando appena inviato
- *<CR><LF>*
- *OK<CR><LF>*

A questo punto è possibile collegarsi alla rete creata dal modulo.

Come si può vedere dalla schermata, quando ci si collega alla rete del server, il modulo invia i seguenti pacchetti:

- *+STA_CONNECTED:"60:ab:67:d3:33:73" <CR><LF>* → indica che un client si è connesso al server, indicandone il suo **MAC address**.
- *+DIST_SPA_IP:"60:ab:67:d3:33:73", "192.168.4.2" <CR><LF>* → indica l'**IP address** del client connesso.

Quando la ESP32 lavora come TCP server, è presente un meccanismo di *time-out*: se il client TCP è connesso al server e non c'è trasmissione di dati per un certo



periodo di tempo, il server si scollega dal client e quindi trasmette il seguente pacchetto:

+STA_DISCONNECTED:"60:ab:67:d3:33:73" <CR><LF>

Il ricollegamento però avviene in automatico: per questo motivo il modulo invierà periodicamente questa sequenza di pacchetti:

+STA_CONNECTED:"60:ab:67:d3:33:73" <CR><LF>
+DIST_SPA_IP:"60:ab:67:d3:33:73", "192.168.4.2" <CR><LF>
+STA_DISCONNECTED:"60:ab:67:d3:33:73" <CR><LF>

Si è scelto di utilizzare un PC come client (si possono usare anche smartphone) e Putty come terminale per l'invio dei dati in wi-fi dal client al server creato dal modulo ESP32. Il modulo inoltra i comandi ricevuti dal terminale alla scheda Renesas, componendo i pacchetti con una struttura specifica che è approfondita nel paragrafo successivo.

3.7 Composizione del pacchetto e comandi ballbot

Monitorando il traffico dati inviato tramite wifi dal terminale PuTTY al modulo ESP si è osservato che i pacchetti contenenti i byte inviati dal terminale (PuTTY) sono inoltrati tramite il canale Tx della porta seriale del modulo, il quale è collegato al canale Rx della scheda, e hanno la seguente struttura:

+IPD,i,n:xxxxxx<CR><LF>

- *i* è l'identificativo del dispositivo connesso in ordine di connessione;
- *n* è il numero di byte di informazione contenuti nel pacchetto ricevuto;
- *xxxxxx* è l'esplicito contenuto informativo del pacchetto;
- <CR> e <LF> sono i caratteri di *Carriage Return* e di *Line Feed*.

È stato quindi necessario definire un formato fisso dei comandi inviati dal terminale in modo tale da avere una struttura del pacchetto univoca ed essere in grado di riconoscerla monitorando il traffico in ricezione sulla porta seriale.

Tale struttura univoca consiste in pacchetti di 3 caratteri (quindi 3 byte, n=3). Inoltre, nonostante il modulo sia stato configurato in modalità “**multiconnection**”, per evitare sovrapposizioni tra i vari dispositivi connessi si è deciso di gestire solo i pacchetti inviati dal primo dispositivo connesso, con identificativo i=0.

```
+IPD,0,3:bON<CR><LF>
<CR><LF>
+IPD,0,2:<CR><LF>
<CR><LF>
```

La struttura dei pacchetti interpretabili dalla scheda, di cui si può vedere un esempio nell'immagine sopra, è quindi la seguente:

+IPD,0,3:xxx<CR><LF>

Per gestire a livello software il riconoscimento di pacchetti di questo tipo è stata definita una struttura **Packet** all'interno del file *wifi_settings.h*, costituita da due stringhe: **header** e **command** (di dimensioni rispettivamente **DIM_HEADER** e **DIM_COMMAND**).

Vengono considerati *header* del pacchetto i primi 8 caratteri e *command* i restanti caratteri fino al <CR>. In base alla convenzione appena illustrata, i comandi riconosciuti come corretti avranno sempre i primi 8 caratteri “+IPD,0,3”, che è stato definito quindi come **CORRECT_HEADER**, e i 3 caratteri dopo ‘:’ e prima di <CR> costituiranno il comando.

Per gestire la divisione in due parti, header e command, di ogni pacchetto, è stato quindi definito lo Splitter ‘:’ dato che costituisce il delimitatore delle due parti del pacchetto.

```

/* Initial commands that the renesas sends for setting the wifi module */
#define WIFI_COMMAND1 "AT+CWMODE=3\r\n"
#define WIFI_COMMAND2 "AT+CIPMUX=1\r\n"
#define WIFI_COMMAND3 "AT+CIPSERVER=1\r\n"

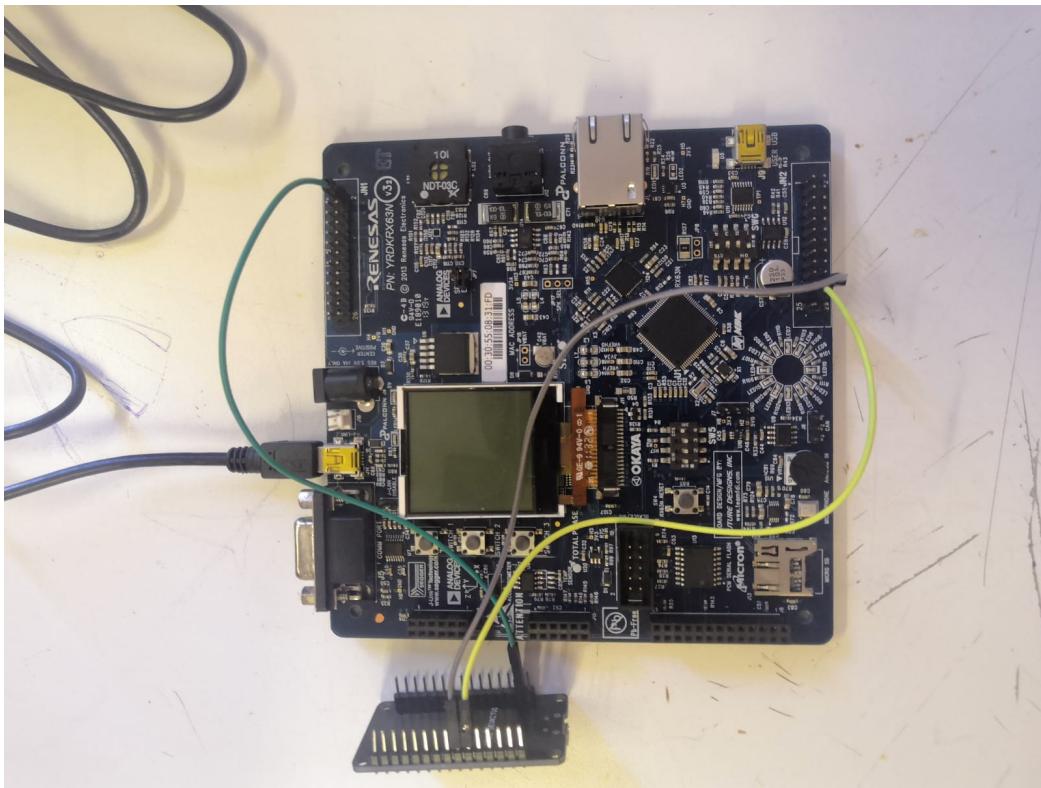
#define SPLITTER ':' /* String splitter between header and command*/
#define DIM_HEADER 8 /* Dimension of the correct command's header */
#define DIM_COMMAND 3 /* Dimension of the correct command */
#define DIM_BUFFER 100 /* Dimension of the data buffer for the communication */
#define CORRECT_HEADER "+IPD,0,3" /* Sample of the correct header of a command */
#define OK_SETTING "OK\r\n\0\0\0\0" /* Ok response from wifi Module to correct command */
#define ERROR_SETTING "ERROR\r\n\0" /* Error response from WiFi module */

typedef struct {
    char header[DIM_HEADER]; /* vector containing the header of the packet */
    char command[DIM_BUFFER]; /* vector containing the command of the packet */
    int n; /* dimension of the command, for a correct command n=DIM_COMMAND */
} packet; /* Struct for managing commands sent by wifi serial communication */

#endif /* WIFI_SETTINGS_H_ */

```

3.8 Trasmissioni e ricezione con modulo WiFi



3.8.1 Impostazione del modulo

Per far avvenire la comunicazione della scheda Renesas con il modulo Wi-Fi si è modificata la funzione *uart_init()* in modo che la scheda inviasse all'avvio i tre

comandi di impostazione iniziale al modulo. Questo è stato implementato mediante il richiamo della funzione *send_data(char* data,int length)*.

I tre comandi sono stati memorizzati all'interno di tre define differenti (*WIFI_COMMAND1*, *WIFI_COMMAND2*, *WIFI_COMMAND3*) all'interno del file *wifi_settings.h*.

```
/* Sending the setting command string to the wifi module */
send_data(WIFI_COMMAND1,str_length(WIFI_COMMAND1));
while (timeout++<=500000){} /*Waiting for response before of sending the next command */
send_data(WIFI_COMMAND2,str_length(WIFI_COMMAND2));
timeout=0;
while (timeout++<=500000){}
send_data(WIFI_COMMAND3,str_length(WIFI_COMMAND3));
```

Tale funzione inserisce la stringa ricevuta come parametro all'interno della coda di trasmissione e abilita l'interrupt di trasmissione così che il dato venga inviato.

```
void send_data(char data[],  unsigned int dim_string)
{
    unsigned int i=0;    /* index for the cycle for inserting data into buffer */
    if (dim_string<(DIM_BUFFER-tx_queue_pointer->counterElements))
        /* if there are sufficients empty slots */
        for (i=0;i<dim_string;i++) /* putting each char of data array into the tx buffer */
    {
        tx_queue_pointer->data[(tx_queue_pointer->head)+i]=data[i]; /* Putting elements from head index */
        tx_queue_pointer->counterElements++;
        tx_queue_pointer->tail++;
        if (tx_queue_pointer->tail>=DIM_BUFFER) /* Managing the tx buffer in a circular way */
            tx_queue_pointer->tail=0;
    }
    interrupt_tx_init();
}
```

L'invio dei tre comandi è stato temporizzato da dei **timeout** utili a non far accavallare fra loro le risposte inviate dal modulo Wi-Fi, per ciascun comando ricevuto.

Se l'impostazione del modulo è andata a buon fine allora per ogni comando trasmesso dalla scheda una delle risposte dal modulo Wi-Fi deve essere la stringa “*OK\r\n*” (quindi la scheda dovrà ricevere per tre volte la suddetta stringa). Tale stringa è stata memorizzata all'interno della *define OK_SETTING* nel file *wifi_settings.h*. Nel caso di errore una delle risposte del modulo Wi-Fi sarà la stringa “*ERROR\r\n*”; tale stringa è stata memorizzata all'interno della *define ERROR_SETTING* del file *wifi_setting.h*.

3.8.2 Elaborazione iniziale dei dati ricevuti dal modulo

Tutti i dati che la scheda riceve dal modulo Wi-Fi sono processati all'interno del file *data_parser.c*: quando l'interrupt di ricezione segnala che la scheda ha ricevuto completamente un pacchetto allora si richiama la funzione *split(char* data)* di questo file, passandogli come parametro il vettore di dati contenuto nella coda di ricezione. Tale funzione restituisce il comando contenuto nel pacchetto se questo è effettivamente un pacchetto contenente un comando o altrimenti la stringa “*noCommand*”.

Come definito precedentemente, il modulo inoltrerà alla scheda i dati ricevuti in Wi-Fi (i comandi) anteponendo un header separato da uno splitter, ‘:’ contenuto nella *define SPLITTER*.

All’interno di *data_parser.c* si utilizza una variabile globale di tipo **packet**. La funzione *split()* ha il compito di individuare il comando ricevuto, separandolo dall’header aggiunto dal modulo Wi-Fi. Si posiziona l’header individuato all’interno del vettore *header* della struttura *packet* e si controlla la correttezza di quest’ultimo prima di inserire il comando nel rispettivo vettore della suddetta struttura (viceversa si farebbero delle computazioni non necessarie nel caso in cui il pacchetto non dovesse contenere un comando).

```
/* Function name: split*/
char* split(char* data)
{
    int i=0;
    int j=0;
    while(data[i] != SPLITTER && i<DIM_HEADER) /* The first DIM_HEADER bits are put in the header
array as soon as the splitter char (':') is found */
    {
        /* The header put in the packet structure may not respect the protocol of the correct header */
        myPacket_pointer->header[i]=data[i];
        i++;
    }
    myPacket_pointer->n=data[i-1]; /* the dimension of the command is the last char of the header */
    i++;

    /* In this phase it is not known if the received packet is a command or not, it is known by checking the header */
    if (parseHeader())
    {

        /* Put DIM_COMMAND bits into the command array, starting from the i bits of the received data */
        while(j<DIM_COMMAND) /* j is the index of the command array in the packet structure */
        {
            myPacket_pointer->command[j]=data[i];
            i++; /* i is the index of the data array */
            j++;
        }

        return parseCommand(); /* parse the command */
    }
    else
        return "noCommand";
}
```

3.8.3 Controllo dell’header ricevuto

Il controllo dell’header del pacchetto è effettuato invocando la funzione *parseHeader()*: questa confronta l’header del pacchetto (contenuto all’interno dell’array *header* della struttura *packet*) con un header campione di un pacchetto contenente un comando dimensionalmente corretto (memorizzato all’interno della *define CORRECT_HEADER* in *wifi_settings.h*). Si utilizza la funzione *strcmp(char* string1,char* string2)* per il confronto: questa restituisce 0 nel caso in cui le due stringhe siano uguali. Il metodo *parseHeader()* restituisce a sua volta il valore 1 (true) nel caso in cui l’header sia corretto, viceversa 0 (false).

```

>int parseHeader()
{
    if (!strcmp(myPacket_pointer->header,ERROR_SETTING))
        /*If the header is equal to ERROR_SETTING string it means that an error has occurred
         while setting the wifi module */
        errorFlag=1;

    if (!errorFlag)    /* if the error flag is on, lcd displays the error */
    {if (!strcmp(myPacket_pointer->header,OK_SETTING))
        lcd_display(LCD_LINE4,"WiFi SET");
    }else
    {
        lcd_display(LCD_LINE4,"WiFi ERROR");
    }

    /* if the header of the packet is not correct the function returns 0 (false) */
    if (!strcmp(myPacket_pointer->header,CORRECT_HEADER))
        /* strcmp returns 0 if the two strings are equal */
        return 1;
    else
        return 0;
}

```

3.8.4 Controllo dell'header per risposte dei comandi AT

Tale funzione individua anche i pacchetti di risposta che il modulo WiFi invia per segnalare che l'impostazione del modulo, attraverso il **comando AT**, precedentemente inviato, è andata a buon fine o meno. Come già discusso precedentemente, il pacchetto conterrà il valore contenuto nella *define OK_SETTING* oppure conterrà il valore contenuto nella *define ERROR_SETTING*. Se l'*header* del pacchetto corrisponde alla stringa *OK_SETTING* si stampa sul display LCD “*WiFi Set*”. Nel caso di errore si attiva una flag (*errorFlag*) che stampa sul display LCD della scritta “*WiFi Error*”. Se il flag di errore viene attivato si stampa sul LCD il warning di errore indipendentemente se i successivi comandi AT inviati hanno un esito di impostazione positivo.

Coma già discusso in precedenza, i pacchetti di risposta ai comandi AT non sono suddivisi in *header* e *command*, indipendentemente da questo si memorizzano i primi **DIM_HEADER** bit ricevuti all'interno del vettore *header* della struttura *packet*. Questa scelta permette di capire dal confronto dell'*header* se l'impostazione del modulo è avvenuta correttamente o no; è una scelta plausibile in quanto le stringhe di corretto/incorrecto settaggio del modulo(*OK_SETTING*, *ERROR_SETTING*) inviate dal modulo stesso hanno una dimensione minore di quella dell'*header* del pacchetto contenente un comando dimensionalmente corretto.

In conclusione, se l'utente leggerà a schermo la scritta “*WiFi Error*” significa che l'impostazione del modulo Wi-Fi non è avvenuta con successo, quindi è consigliato resettare il modulo e successivamente resettare la scheda per ripetere l'invio dei comandi AT di impostazione iniziale. La funzione *parseHeader()* restituirà il valore 0 quando si ricevono i pacchetti di risposta ai comandi AT.

3.8.5 Parsing del comando

Nel caso in cui il pacchetto contenga un comando dimensionalmente corretto, la funzione *parseHeader()* restituisce il valore 1 e si procede quindi con l'inserimento del comando nel vettore *comand* della struttura *packet*.

A questo punto si richiama la funzione *parseCommand()* che confronta il comando ricevuto con i comandi disponibili, definiti all'interno delle *define* nel file *ballbot_command.h*, e stampa sul display la funzione da eseguire.

```
char* parseCommand()
{
    if (!strcmp(myPacket_pointer->command,START))
        lcd_display(LCD_LINE8,"START");
    if (!strcmp(myPacket_pointer->command,STOP))
        lcd_display(LCD_LINE8,"STOP");
    if (!strcmp(myPacket_pointer->command,FORWARD))
        lcd_display(LCD_LINE8,"Forward");
    if (!strcmp(myPacket_pointer->command,BACKWARD))
        lcd_display(LCD_LINE8,"Backward");
    if (!strcmp(myPacket_pointer->command,LEFT))
        lcd_display(LCD_LINE8,"Left");
    if (!strcmp(myPacket_pointer->command,RIGHT))
        lcd_display(LCD_LINE8,"Right");

    return myPacket_pointer->command;
}

/* Received command for ballbot function START,STOP,FORWARD,BACKWARD,LEFT,RIGHT*/
#define START "bON"
#define STOP "bOF"
#define FORWARD "bAV"
#define BACKWARD "bIN"
#define LEFT "bSX"
#define RIGHT "bDX"
```

Il comando viene restituito alla primaria funzione chiamante all'interno dell'interrupt di ricezione nel file *ballbot_uart.c*. In questo punto si potrebbero richiamare delle funzioni che, prendendo in ingresso il comando ricevuto, effettuano praticamente quella funzione (ad esempio movimento dei motori per far spostare il ballbot in avanti).

Dopo l'elaborazione il comando viene pulito mediante il metodo *clearCommand()* all'interno di *data_parser.c*, così da essere pronti per una successiva ricezione.

```
void clearCommand()
{
    /* if the command vector has been loaded, it means that the received packet contained a
     * correct command, so its dimension is of DIM_COMMAND
     */
    int i;
    for (i=0;i<DIM_COMMAND;i++) /* clear each bit of the command vector of the packet */

    {
        myPacket_pointer->command[i]=0;
    }
}
```

4. Appendice

ProgettoLaboratorio.c

```
#include <stdbool.h>
#include <stdio.h>
#include <machine.h>
#include "platform.h"
#include "ballbot_uart.h"

/***** Function name : main *****
* Description   : Start the program
* Arguments     : none
* Return Value  : none
*****/
void main(void)
{
    lcd_initialize();

    /* Clear LCD */
    lcd_clear();

    /* Display message on LCD */
    lcd_display(LCD_LINE1,"BallBot");
    lcd_display(LCD_LINE2,"WiFi Serial");
    lcd_display(LCD_LINE6,"Command:");

    uart_init(); /* Initialize the uart communication */

}
```

data_parser.h

```
#ifndef DATA_PARSER_H_
#define DATA_PARSER_H_

/***** Prototypes for exported functions *****/
Prototypes for exported functions
*****/
```

```

int parseHeader();
char* parseCommand();
char* split(char*);
void clearCommand();

#endif

```

ballbort_uart.h

```

#ifndef BALLBOT_UART_H_
#define BALLBOT_UART_H_
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <machine.h>
#include "platform.h"
#include "data_parser.h"
#include "wifi_settings.h"

*****  

*****  

Macro definitions  

*****  

*****/  

/* struct used for managing the data of the communication */
typedef struct {
    unsigned char data [DIM_BUFFER]; /* chars buffer used with a
circular nature */
    unsigned int head; /* pointer to the first item of the buffer */
    unsigned int tail; /* pointer to the last item of the buffer */
    unsigned int counterElements;
}queue;  

*****  

*****  

Prototypes for exported functions  

*****  

*****/  

void uart_init();
void interrupt_tx_init();
void interrupt_tx_disable();
void interrupt_rx_init();
void interrupt_rx_disable();
void rx_queue_init();
void tx_queue_init();
void send_data(char[],unsigned int);

```

```

int queue_is_empty();
int queue_is_full();
void queue_clear(queue*);

#endif

```

ballbot_command.h

```

#ifndef BALLBOT_COMMAND_H_
#define BALLBOT_COMMAND_H_

/* Received command for ballbot function
START,STOP,FORWARD,BACKWARD,LEFT,RIGHT*/
#define START "bON"
#define STOP "bOF"
#define FORWARD "bAV"
#define BACKWARD "bIN"
#define LEFT "bSX"
#define RIGHT "bDX"

#endif /* BALLBOT_COMMAND_H_ */

```

wifi_settings.h

```

#ifndef WIFI_SETTINGS_H_
#define WIFI_SETTINGS_H_

*****  

*****  

Macro definitions  

*****  

*****/  

/* Initial commands that the renesas sends for setting the wifi module  

*/
#define WIFI_COMMAND1 "AT+CWMODE=3\r\n"
#define WIFI_COMMAND2 "AT+CIPMUX=1\r\n"
#define WIFI_COMMAND3 "AT+CIPSERVER=1\r\n"

#define SPLITTER ':' /* String splitter between header and command*/
#define DIM_HEADER 8 /* Dimension of the correct command's header */
#define DIM_COMMAND 3 /* Dimension of the correct command */
#define DIM_BUFFER 100 /* Dimension of the data buffer for the
communication */
#define CORRECT_HEADER "+IPD,0,3" /* Sample of the correct header of a
command */

```

```

#define OK_SETTING "OK\r\n\0\0\0\0" /* Ok response from wifi Module to
correct command */
#define ERROR_SETTING "ERROR\r\n\0" /* Erroro response from WiFi module
*/
typedef struct {
    char header[DIM_HEADER];      /* vector containing the header of the
packet */
    char command[DIM_BUFFER];     /* vector containing the command of the
packet */
    int n; /* dimension of the command, for a correct command
n=DIM_COMMAND */
} packet; /* Struct for managing commands sent by wifi serial
communication */

#endif /* WIFI_SETTINGS_H_ */

```

ballbot_uart.c

```

#include "ballbot_uart.h"

*****
*****  

Private global variables  

*****  

/* Rx and Tx queue element used for managing the reception data buffer
and transmission data buffer */  

queue rx_queue;  

queue tx_queue;  

/* Rx and Tx queue pointers */  

queue* tx_queue_pointer;  

queue* rx_queue_pointer;  

*****
*****  

* Function name : interrupt_tx_init  

* Description   : Initialize the TXI8 interrupt of SCI8  

* Arguments     : none  

* Return Value  : none  

*****  

void interrupt_tx_init()  

{

```

```

        IR(SCI12,TXI12)=1; /* Enable the interrupt in the SCI12 peripheral
*/
        IEN(SCI12,TXI12)=1;
}

/*********************  

*****  

* Function name : interrupt_rx_init  

* Description   : Initialize the RXI8 interrupt of SCI8  

* Arguments     : none  

* Return Value  : none  

*****  

void interrupt_rx_init()
{

    IEN(SCI12,RXI12)=1; /* Enable the interrupt in the ICU */
}

/*********************  

*****  

* Function name : rx_queue_init  

* Description   : Initialize the rx queue setting the pointer to the
head and the tail of the
* rx data buffer
* Arguments     : none
* Return Value  : none
*****  

void rx_queue_init()
{
    rx_queue_pointer=&rx_queue;
    rx_queue_pointer->head=0;
    rx_queue_pointer->tail=0;
    rx_queue_pointer->counterElements=0;
    /* Empty queue conditions: head points to the same element pointed
by tail and counterElements=0 */
}

/*********************  

*****  

* Function name : tx_queue_init  

* Description   : Initialize the tx queue setting the pointer to the
head and the tail of the
* tx data buffer
* Arguments     : none
* Return Value  : none
*****  

void tx_queue_init()
{
    tx_queue_pointer=&tx_queue;
}

```

```

    tx_queue_pointer->head=0;
    tx_queue_pointer->tail=0;
    tx_queue_pointer->counterElements=0;
    /* Empty queue conditions: head points to the same element pointed
    by tail and counterElements=0 */
}

/***** *****
* Function name : uart_init
* Description   : Set up all the registers for enabling UART
communication
* Arguments     : none
* Return Value  : none
***** */
void uart_init()
{
    int timeout=0;

    /* Enable the UART peripheral to operate */
    /* (the peripheral must be enabled before being used).*/
    /* The SCI12 channel is used because the TxD12 and RxD12
register are
       * accessible by JN2's pins. (ref. Hardware Manual Chapt.22
page 703)*/
    /* To enable SCId (c=0..12) */
    /* the register MSTP(SCId) must be activated, because*/
    /* this register is protected on writing by PRCR protection
register, */
    /* before activate MSTP(SCId) it is necessary:*/
    /* a) to set off the PRCR, */
    /* b) enable desired SCId SCId (c=0..12) channel (in this
case SCI12, c=12)*/
    /* c) to set on the PRCR. */

    #ifdef PLATFORM_BOARD_RDKRX63N
        SYSTEM.PRCR.WORD = 0xA50B; /* Protect off */
    #endif

    MSTP(SCI12)=0; /* cancel stop state of SCI12 peripheral to
enable writing to it */
    /* for SCI12 PE1 is TxD12 and PE2 is RxD12 */
    #ifdef PLATFORM_BOARD_RDKRX63N
        SYSTEM.PRCR.WORD=0xA500; /* Protect off*/
    #endif

    /* Protect set on (ref. Hardware Manual Chapt.13.1.1)*/
    /* write A5 (in hexadecimal)
to the eight higher-order */
    /* bits and 0B (in
hexadecimal) to the eight lower-order */
}

```

```

        /* where B in hexadecimal is
equivalent to 1011 in Binary */
        /* therefore it sets PRC3,
PRC1 and PRC0 to 1 */

#endif

/* Set up the UART I/O port and pins */

//MPC.PE2PFS.BIT.ISEL=1;      /*alternative code: ISEL is
the interrupt (ref. Hardware Manual Chapt.22.2.16 pag 734)*/
//MPC.PE2PFS.BIT.PSEL=0xC;   /*This value of PSEL is for
RxD12 */
MPC.PE2PFS.BYTE=0x4C;    /*P23 of JN2 is RxD12*/
MPC.PE1PFS.BYTE=0x4C;    /*P22 of JN2 TxD12*/

PORTE.PDR.BIT.B1=1;    /* TxD12 is output */
PORTE.PDR.BIT.B2=0;    /* RxD12 is input */
PORTE.PMR.BIT.B1=1;    /*TxD12 is peripheral */
PORTE.PMR.BIT.B2=1;    /* RxD12 is a peripheral */

/* Set data transfer format in Serial Mode Register (SMR)
- HW 35.2.5
    * -Asynchronous Mode
    * -8 bits
    * -no parity
    * -1 stop bit
    * PCLK clock (n=0)
    */
SCI12.SMR.BYTE=0x00;
SCI12.SCMR.BIT.SMIF=0; /* Set to 0 for serial
communications interface mode */
SCI12.BRR=48000000 / ((64/2) * 115200) -1; /*
    *The baud rate chosen is 115200 equals to the baud rate of
the wifi module é/

    /* Clear IR bits (of the interrupt) for SCI12 registers:
TXI, RXI, TEI */
    IR(SCI12,RXI12)=0; //Clear any pending ISR
    IR(SCI12,TXI12)=0;
    IR(SCI12,TEI12)=0;

    IPR(SCI12,RXI12)=4; /* Set interrupt priority, not using the
transmit end interrupt */
    IPR(SCI12,TXI12)=4;

    /* Disable all the interrupt (TXI,RXI,TEI) */
    IEN(SCI12,TXI12)=0;
    IEN(SCI12,RXI12)=0;
    IEN(SCI12,TEI12)=0;

```

```

        /* Clear SCR register: clear bits TIE, RE and TEIE in SCR
to 0. Set CKE to internal */
        SCI12.SCR.BYTE=0x00;
        /* Enable RXI and TXI interrupts in SCI peripheral - (ref.
Hardware Manual cp. 35.2.6 */
        SCI12.SCR.BIT.RIE=1;      //Reception interrupt request
(RXI12) is enabled setting the bit to 1
        SCI12.SCR.BIT.TIE=1;      //Transmission interrupt
request (TXI12) is enabled setting the bit to 1
        SCI12.SCR.BIT.TEIE=0;      //A TEI (Transmit End
Interrupt) interrupt request is disabled, not used
        SCI12.SCR.BIT.TE=1;      //Serial transmission is enabled
setting bit to 1
        SCI12.SCR.BIT.RE=1;      //Serial reception is enabled
setting bit to 1

        /* Initializing the Tx and Rx queues */
        rx_queue_init();
        tx_queue_init();

        /* Sending the setting command string to the wifi module */
        send_data(WIFI_COMMAND1,str_length(WIFI_COMMAND1));
        while (timeout++<=500000){} /*Waiting for response before
of sending the next command */
        send_data(WIFI_COMMAND2,str_length(WIFI_COMMAND2));
        timeout=0;
        while (timeout++<=500000){}
        send_data(WIFI_COMMAND3,str_length(WIFI_COMMAND3));

        LED15=LED_ON;    /* Led 15 means that the uart init has been
completed */

        /* Calling the initialization of interrupt for the
communication */
        interrupt_rx_init();
}

/*********************************************
*****
* Function name: SCI8_TXI8_int
* Description : Interrupt Service Routine for TxD8. It is fired when
the TSR register of SCI8 is empty.
* Argument     : none
* Return value : none

```

```

*****
*****/
#pragma interrupt SCI12_TXI12_int (vect = VECT_SCI12_TXI12, enable)
static void SCI12_TXI12_int()
{

    if (!queue_is_empty(tx_queue_pointer)) /* if the tx queue is not
empty */
    {

        SCI12.TDR=tx_queue_pointer->data[tx_queue_pointer->head]; /*
putting into the data buffer
            register (TDR) of the SCI8 channel the first element of the
data buffer trasmission */

        tx_queue_pointer->head++; /* Now the head of the buffer will
be the next element */
        tx_queue_pointer->counterElements--;
        if (tx_queue_pointer->counterElements==0) /* Disable the tx
interrupt if all the elements have been transmitted */
            {interrupt_tx_disable();
            queue_clear(tx_queue_pointer); /* Clear the tx queue after
the transmission */
        }

    }
}

/*****
*****
* Function name: SCI8_RXI8_int
* Description : Interrupt Service Routine for RXD8. It is fired when
received data are stored in RDR
* Argument     : none
* Return value : none
*****
****/
#pragma interrupt SCI12_RXI12_int (vect = VECT_SCI12_RXI12, enable)
static void SCI12_RXI12_int()
{

    int timeout=0;

    /* Frame and Parity Error handling */
    while(SCI12.SSR.BIT.ORER && timeout++<=10000)
    {
        SCI12.SSR.BIT.ORER=0;
    }

    if (SCI12.SSR.BIT.FER)
        SCI12.SSR.BIT.FER=0;
    if (SCI12.SSR.BIT.PER)

```

```

SCI12.SSR.BIT.PER=0;

    uint8_t read_byte=SCI12.RDR; /* Read byte from RDR register */
    /* Putting the data from RDR register into the reception buffer */
    rx_queue_pointer->data[rx_queue_pointer->tail]= read_byte;
    rx_queue_pointer->counterElements++; /* Incrementing the
counter of received elements */
    rx_queue_pointer->tail++; /* Incrementing the tail pointer to the
next position (it will point to an empty element) */

    /* if the rx queue is full or last element of the rx queue is the
'\n' we can start processing the received packet */
    if (queue_is_full(rx_queue_pointer) || rx_queue_pointer-
>data[rx_queue_pointer->tail-1]=='\n') /* The wifi module sends the
received command

to renesas with '\n' in the final position */
{
    char command[DIM_COMMAND]=split(rx_queue_pointer); /*
splitting the received data from wifi module */
    clearCommand(); /* clear the command array for being ready for
the next communication */
    /* it is not necessary to clear the header because it is
overwritten every time that a packet (correct or incorrect) is received
*/
    interrupt_rx_disable(); /* Disabling the rx interrupt in order
to clear the rx queue completely without the risk of getting other
packets */
    queue_clear(rx_queue_pointer);
}

/* Enabling again the rx interrupt for being ready for receiving
the next packet */
interrupt_rx_init();

}

/*****
* Function name : queue_is_full
* Description   : Check if a queue is full
* Arguments     : queue pointer to check
* Return Value  : int value --> 1 the queue is full, 0 the queue is not
full
*****
/int queue_is_full(queue* queue_pointer)
{
    if (queue_pointer->counterElements>=DIM_BUFFER) /* If there is a
number of elements equal to the buffer's dimension

```

```

        the buffer is full
*/
    return 1;
else
    return 0;
}

/***** Function name : queue_is_empty *****
* Description   : Check if a queue is empty
* Arguments     : queue pointer to check
* Return Value  : int value --> 1 the queue is empty, 0 the queue is
not empty
*****/
int queue_is_empty(queue* queue_pointer)
{
    /*The queue is empty when countElements is 0 */
    if (!queue_pointer->counterElements)
        return 1;
    else
        return 0;
}

```

R

```

/***** Function name : send_data *****
* Description   : Send data from P21 of JN1's renesas periphal
* Arguments     : char array to transmit and its dimension
* Return Value  : none
*****/
void send_data(char data[],  unsigned int dim_string)
{

    unsigned int i=0;    /* index for the cycle for inserting data into
buffer */
    if (dim_string<(DIM_BUFFER-tx_queue_pointer->counterElements)) /* if there are sufficients empty slots */
        for (i=0;i<dim_string;i++) /* putting each char of data array
into the tx buffer */
    {
        tx_queue_pointer->data[(tx_queue_pointer-
>head)+i]=data[i]; /* Putting elements from head index */
        tx_queue_pointer->counterElements++;
        tx_queue_pointer->tail++;
        if (tx_queue_pointer->tail>=DIM_BUFFER) /* Managing the
tx buffer in a circular way */

```

```

        tx_queue_pointer->tail=0;
    }
    interrupt_tx_init();
}

/*********************  

*****  

* Function name : interrupt_tx_disable  

* Description   : Disable TXI8 interrupt of SCI8  

* Arguments     :  

* Return Value  :  

*****  

*****/  

void interrupt_tx_disable()  

{  

    IR(SCI12,TXI12)=0;  

    IEN(SCI12,TXI12)=0;  

}

/*********************  

*****  

* Function name : interrupt_rx_disable  

* Description   : Disable RXI8 interrupt of SCI8  

* Arguments     :  

* Return Value  :  

*****  

*****/  

void interrupt_rx_disable()  

{  

    IEN(SCI12,RXI12)=0;  

}

/*********************  

*****  

* Function name : str_length  

* Description   : Returns the length of the input string  

* Arguments     : String  

* Return Value  : int value -> Length of the string  

*****  

*****/  

int str_length(char string[])
{
    unsigned int i=0;
    while(i<DIM_BUFFER)
    {
        if(string[i]==0)
            return i;
        else
            i++;
    }
    return DIM_BUFFER;
}

```

```

*****
* Function name : queue_clear
* Description   : Clears the buffer of the queue
* Arguments     : queue*
* Return Value  : none
*****
*****/
void queue_clear(queue* queue_pointer)
{
    int i;
    for(i=0; i<queue_pointer->tail; i++) /* Clear all the bits of the
queue */
    {
        queue_pointer->data[i]=0;
    }
    queue_pointer->tail=0;
    queue_pointer->head=0;
    queue_pointer->counterElements=0;
}

```

data_parser.c

```

#include "data_parser.h"
#include <stdbool.h>
#include <stdio.h>
#include <machine.h>
#include "platform.h"
#include "wifi_settings.h"
#include "ballbot_command.h"

packet myPacket; /* Object that contains the received packet, including
its header and data */
packet* myPacket_pointer=&myPacket; /* Pointer to the packet object */
int errorFlag=0; /* Flag for checking error setting of wifi module */
*****
***** 
* Function name: parseHeader
* Description   : Compare the header of the received packet with the
CORRECT_HEADER in order to find a correct packet
* Argument      : none
* Return value  : 1 if the packet is correct command packet, 0 if it is
not a command packet
*****
*****/
int parseHeader()
{

```

```

    if (!strcmp(myPacket_pointer->header,ERROR_SETTING)) /*If the
header is equal to ERROR_SETTING string it means that an error has
occured
                                while
setting the wifi module */
        errorFlag=1;

    if (!errorFlag)      /* if the error flag is on, lcd displays the
error */
        {if (!strcmp(myPacket_pointer->header,OK_SETTING))
            lcd_display(LCD_LINE4,"WiFi SET");
        }else
        {
            lcd_display(LCD_LINE4,"WiFi ERROR");
        }

    /* if the header of the packet is not correct the function returns
0 (false) */
    if (!strcmp(myPacket_pointer->header,CORRECT_HEADER)) /* strcmp
returns 0 if the two strings are equal */
        return 1;
    else
        return 0;
}

/*********************************************
*****
* Function name: clearCommand
* Description : Clear the command array of the packet structure after
its use
* Argument     : none
* Return value : none
*****
*/
void clearCommand()
{
    /* if the command vector has been loaded, it means that the
received packet contained a
     * correct command, so its dimension is of DIM_COMMAND
     */
    int i;
    for (i=0;i<DIM_COMMAND;i++) /* clear each bit of the command vector
of the packet */

    {
        myPacket_pointer->command[i]=0;
    }
}

/*********************************************
*****
* Function name: parseCommand

```

```

* Description : Parse the received command
* Argument     : none
* Return value : Char* pointing to the parsed command
*****
*****/
char* parseCommand()
{
    if (!strcmp(myPacket_pointer->command,START))
        lcd_display(LCD_LINE8,"START");
    if (!strcmp(myPacket_pointer->command,STOP))
        lcd_display(LCD_LINE8,"STOP");
    if (!strcmp(myPacket_pointer->command,FORWARD))
        lcd_display(LCD_LINE8,"Forward");
    if (!strcmp(myPacket_pointer->command,BACKWARD))
        lcd_display(LCD_LINE8,"Backward");
    if (!strcmp(myPacket_pointer->command,LEFT))
        lcd_display(LCD_LINE8,"Left");
    if (!strcmp(myPacket_pointer->command,RIGHT))
        lcd_display(LCD_LINE8,"Right");

    return myPacket_pointer->command;
}

*****
*****/
* Function name: split
* Description : Split the received packet in header and command
* Argument     : none
* Return value : Char* pointing to the parsed command
*****
*****/
char* split(char* data)
{
    int i=0;
    int j=0;
    while(data[i] != SPLITTER && i<DIM_HEADER) /* The first DIM_HEADER
bits are put in the header array as soon as the splitter char ':' is
found */
    {
        /* The header put in the packet structure may not respect the
protocol of the correct header */
        myPacket_pointer->header[i]=data[i];
        i++;
    }
    myPacket_pointer->n=data[i-1]; /* the dimension of the command is
the last char of the header */
    i++;

    /* In this phase it is not knowned if the received packet is a
command or not, it is knowned by checking the header */
}

```

```

if (parseHeader())
{

    /* Put DIM_COMMAND bits into the command array, starting from
the i bits of the received data */
    while(j<DIM_COMMAND)      /* j is the index of the command array
in the packet structure */
    {
        myPacket_pointer->command[j]=data[i];
        i++;      /* i is the index of the data array */
        j++;
    }

    return parseCommand();      /* parse the command */

} else
    return "noCommand";
}

```

5. Conclusioni

*“La comunicazione non è quello che diciamo, bensì
quello che arriva agli altri.”*

(Thorsten Havener)

Lo scopo del progetto è stato quindi raggiunto: siamo in grado di trasmettere brevi stringhe da un controller remoto alla scheda Renesas.

Per raggiungere il nostro obiettivo sono stati presi in considerazione i 6 comandi principali (avanti, indietro, destra, sinistra, acceso, spento), nel caso in cui sia necessario l'analisi di nuovi tipi di comandi è sufficiente ampliare la gamma di opzioni presenti nel parseCommand(). Essendo possibile trasmettere tutti i 96 caratteri codificabili in ASCII, le disposizioni totali possibili di 3 caratteri risultano all'incirca 884.736 (vanno poi escluse alcuni casi particolari come “000” che mandano in errore il parser) che sono molto più che sufficienti per prendere in considerazione le possibili richieste dall'utente alla scheda.

Alla ricezione di un comando si è voluto stampare a video un messaggio di ritorno, questo ha lo scopo puramente dimostrativo che la catena di trasmissione-ricezione-analisi è andata a buon fine.

Lo sviluppo delle conseguenze e degli effetti prodotti dalla scheda è lasciata ad altri colleghi, che implementando il nostro progetto saranno in grado di considerare la richiesta di un comando da parte dell'utente da un controller remoto come una semplice ricezione di una stringa in input sulla scheda.

6. Bibliografia

- Elisabetta Cipriani, Prof. Massimiliano Pirani, Prof. Andrea Bonci, Infrastruttura di controllo Wi-Fi per rete di veicoli autonomi con elettronica Renesas serie RX, a.a 2013-2014.
- Prof. Andrea Bonci, 08a_Comunicazione Seriale Teoria.pdf
- https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf
- <https://atadiat.com/en/e-esp8266-esp32-wifi-serial-bridge-converter/>
- <https://www.politesi.polimi.it/bitstream/10589/29601/4/tesi.pdf>
- https://www.renesas.com/sg/en/doc/products/tools/r20ut2532eu0100_yrd_krx63n_um.pdf
- https://www.renesas.com/eu/en/doc/products/mpumcu/doc/rx_family/r01_uh0041ej0180_rx63n631.pdf?key=1ee8ec74f1dc8703f8f540f54e33d33c