Giuliano Pellegrini Parisi

Politecnico di Torino Mat. s203675

Anno accademico 2016/2017

Relatore Prof.re Flavio Canavero

Tesi di Laurea Magistrale in Ingegneria Informatica

Realizzazione stazione meteo con salvataggio e visualizzazione dati nel Cloud

Guida pratica

alla programmazione

di Raspberry 3 con C#

Realizzazione stazione meteo con salvataggio e visualizzazione dati nel Cloud

Guida pratica

alla programmazione

di Raspberry 3 con C#

Anno accademico 2016/2017

Laurea Magistrale in Ingegneria Informatica

Relatore

Prof.re Flavio Canavero

Studente

Giuliano Pellegrini Parisi

Indice

Premessa

Il progetto "Stazione meteo" con Raspberry Pi 3 ha lo scopo di evidenziare le possibilità offerte da un sistema embedded nell'automatizzare, in modo semplice, la gestione di un rilevatore di velocità e direzione del vento LaCross TX20, tramite l'ambiente di sviluppo Microsoft Visual Studio 2015.

Il linguaggio di programmazione scelto è il C#, il quale verrà impiegato per lavorare direttamente sull'interfaccia GPIO della scheda in modo chiaro e semplice, ma al tempo stesso permettendo di sfruttare le potenzialità del Cloud sia in termini di salvataggio delle informazione che visualizzazione dei dati in formato report.

La realizzazione del progetto avviene in modo lineare, dal semplice testing delle porte della scheda Raspberry, col fine di testarne il funzionamento, fino al collaudo finale dell'anemometro con lettura dei dati dal Cloud tramite browser internet.

Questa relazione ha l'obiettivo di marcare gli aspetti più importanti, sia in termini di interfacciamento tramite breadboard, sia di realizzazione del software evidenziando in modo chiaro e preciso le scelte fatte dal progettista.

Un sistema embedded offre oggi giorno moltissime possibilità nel campo dell'automazione, non per nulla in molti testi il progettista viene chiamato Maker, colui crea ed utilizza questi fantastici dispositivi per realizzare i più stravaganti progetti. In questa trattazione ovviamente non è possibile vedere fino a che punto un sistema embedded può essere messo "alle corde", visto che il più delle volte il Maker è messo alla prova non dal Raspberry, ma dal dispositivo che vuole pilotare, ossia deve capire in che modo interfacciarsi da un punto di vista elettrico, in che modo posizionare i componenti elettronici per leggere/scrivere da e verso il dispositivo, quale protocollo utilizzare o, peggio ancora, effettuare il reverse engineering per capire il protocollo di comunicazione, come avvenuto in fase di testing dell'anemometro TX20 in laboratorio sfruttando un oscilloscopio digitale Rigol DS1052E.

Giuliano Pellegrini Parisi

Capitolo 1

Raspberry Pi 3

Questo capitolo descrive le caratteristiche fondamentali di un sistema embedded. Impareremo le differenze tra le tecnologie presenti sul mercato, col fine di trovare la soluzione hardware più consona alle nostre esigenze.

1.1 *I sistemi embedded*

Il mondo dell'elettronica digitale si è voluto molto rapidamente in questo ultimo decennio, producendo integrati con elevato fattore di integrazione che ha permesso di raccogliere in poco spazio potenzialità che, fino al secolo scorso, obbligavano i costruttori di schede elettroniche PCB, ha difficili lavori di progettazione, di compatibilità e, per gli addetti ai lavori, anche di manutenzione in caso di guasti.

1.1.1 Le tipologie di sistemi embedded

Sul mercato oggi giorno esistono una moltitudine di schede e sistemi embedded, ma con caratteristiche diverse, sia in termini di potenza di calcolo, sia in termini di consumo energetico. Una notevole varietà di schede embedded è reperibile sul mercato cinese, dove è facile trovare prodotti della HAOYU Electronics che propone, a prezzi modesti, delle schede HY-LandTiger, le quali sono sprovviste di sistema operativo, ma sfruttano una memoria flash per caricare direttamente all'avvio elettrico del device il programma in linguaggio macchina, che il programmatore ha precedentemente realizzato, compilato e copiato nella flash. Il linguaggio utilizzato in questa tipologia di dispositivi è il C, il quale

obbliga il programmatore a gestire tutto, dall'allocazione della memoria, fino all'accesso a basso livello dei dispositivi presenti sulla scheda tramite Assembly. Questo aspetto è sicuramente poco accattivante per coloro che sono alla "prime armi", visto che è necessario possedere un background non solo nella scrittura di codice in C, ma anche nella scrittura di codice/algoritmi di basso livello con l'ovvia conseguenza che non è pensabile di utilizzare Assembly senza conoscere molto bene il datasheet di tutta la scheda e i componenti hardware installati. Una interruzione hardware impostata dal programmatore in modo errato, oltre a non funzionare, potrebbe provocare effetti poco desiderati.

Il vantaggio nell'impiego di una tecnologia HY-LandTiger, è nella velocità di esecuzione del processo il quale, grazie all'assenza del sistema operativo, è l'unico codice che utilizza il processore e la memoria ram in modo esclusivo. Il programmatore può quindi, tramite l'ambiente di sviluppo, configurare il linker per avere un codice macchina con istruzioni assolute e definire a priori l'indirizzo fisico nella locazione di memoria ram. Il termine scheda embedded può essere visto come device privo di sistema operativo che permette l'esecuzione immediata, in modalità esclusiva, dell'unico processo presente nella memoria flash.

La scelta di una HY-LandTiger deve essere fatta anche sulla documentazione reperibile in rete che al momento è alquanto limitata.

Un'altra scheda embedded, è la famosissima Arduino e varianti, la quale anch'essa è priva di sistema operativo e permette la scrittura di codice C-style in modo molto più semplice e molto più veloce rispetto alla HY-LandTiger. Le conoscenze richieste di elettronica sono sempre consigliate, ma nettamente inferiori rispetto alla concorrente cinese, così come il costo che si rivela quasi dimezzato. Nei paragrafi successivi verranno elencate le caratteristiche di tali schede.

Un sistema embedded può essere considerato come un'evoluzione di una scheda embedded, con la differenza cardine che permette l'installazione di un sistema operativo quali Linux o Windows 10 IoT, quest'ultimo solo per determinate tecnologie.

Il device Raspberry Pi 2 e versione 3 possono ospitare molte distribuzioni Linux, quali Raspbian, Arch Linux, Pidora e molte altre, ecco perché sovente si tende ad utilizzare il pacchetto NOOBS che permette all'utente di scegliere quale distribuzione Linux utilizzare per l'installazione. Un'altra tecnologia embedded molto impiegata in USA è la BBB ossia BeagleBone Black che sfrutta le potenzialità di una distribuzione Angstrom Linux. La breve

carrellata di dispositivi deve porre un quesito che è alla base delle scelta tecnologica per la stazione meteo, ossia:

<< Serve il sistema operativo? >>

L'obiettivo che si vuole raggiungere è una stazione di rilevamento del vento che in modo indipendente possa inviare sul Cloud i propri dati, così che l'utenza di rete possa avere accesso in formato grafico alle informazioni. La stazione deve poi essere scalabile, ossia permettere l'inserimento di nuovi componenti, quali ad esempio sensori per il rilevamento della temperatura istantanea, ma anche sensori di pressione atmosferica. L'inserimento di nuova sensoristica è sempre fonte di intervento manuale, ma che potrebbe essere minimo, se tutti i sensori vengono installati a bordo di una scheda PCB (Printed Circuit Board), la quale una volta inserita nel box della stazione, può venire correttamente alimentata e collegata all'interfaccia GPIO, così che il programmatore da remoto possa apportare le dovute modifiche al software che gira a bordo della scheda. Progettare e/o acquistare un box ad hoc per tale esigenze è un fattore che il Maker deve considerare come requisito in fase di scelte progettuali.
Sulla base di quanto indicato in precedenza, è chiaro che vi deve essere un accesso da remoto per aggiornare il software, vi deve essere la possibilità di integrazione del software con il mondo Cloud, motivi che spingono a optare per un sistema embedded che possa offrire tutte queste possibilità. La scelta fatta dall'autore è l'impiego della nuova scheda Raspberry Pi 3 (al momento in cui si scrive) con l'ausilio del sistema operativo di casa Redmond, Microsoft Windows 10 IoT (Internet of Things) visto che si vuole programmare in linguaggio ad oggetti C# tramite l'ambiente di sviluppo Visual Studio 2015, sfruttando il framework .NET.
E' importante evidenziare che l'intero progetto è facilmente esportabile anche in Linux, visto che C# sotto Linux può sfruttare non il framework .NET, il quale è copyright Microsoft, ma il framework Mono rilasciato sotto licenza GPL, permettendo quindi di eseguire codice gestito C# tramite il CLR (Common Language Runtime).
La scelta di un linguaggio diverso dal C# come Python è anch'essa molto allettevole, visto la portabilità degli script Python e la conseguente facilità di esecuzione in ambiente Linux. Sicuramente un buon programmatore si trova a più agio quando può utilizzare, per la

scrittura del proprio programma, un ambiente di sviluppo potente e collaudato, in questo Microsoft Visual Studio non ha rivali.

Il costo di un Raspberry Pi 3 non è elevatissimo, anche se il prezzo lievita nel momento in cui si procede all'acquisto di sensoristica o di hardware generico esterno. Dipende sempre da quello che si deve fare. La Tab.1 sottostante riporta una bozza dei prezzi delle varie tecnologie che può variare anche del 20%-30% se acquistato direttamente sul mercato cinese.

Nome prodotto	Prezzo
Raspberry Pi 2 modello B	€ 35,00
Raspberry Pi 3	€ 45,00
Arduino Uno R3 ATMega 328P	€ 6,00
BeagleBone Black	€ 75,00
HY-LandTiger	€ 55,00
Intel Galileo Gen 2	€ 100,00

Tab.1

I prezzi sono alquanti differenti proprio per le differenti caratteristiche hardware, in primis il processore multi core, la memoria e la sensoristica a bordo del PCB.

Un fattore molto importante a cui si deve fare attenzione è l'ambiente di lavoro in cui è installato il device, visto che ambienti industriali potrebbero essere soggetti a temperature elevate, soprattutto nei mesi estivi, ecco quindi la necessità di optare per dei sistemi orientati all'industria, quali gli SBC (Single Board Computer). Esistono in commercio una varietà di vendor che producono degli SBC, dal modello SBC-GX533 all'Aludra Atom, dal Vector Atom all'Antares i7, dal Catalyst EC Atom al Gemini Dual Core e moti altri. Un riferimento molto esaustivo è reperibile alla pagina http://www.eurotech.com/en/products.

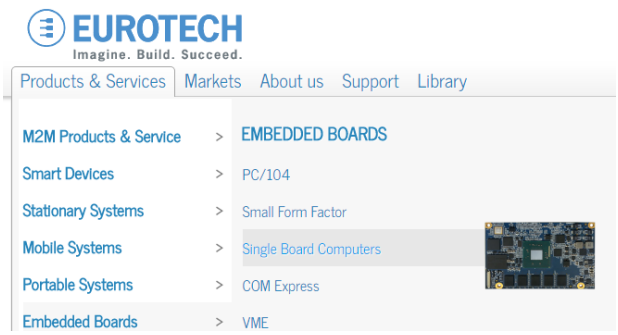


Fig.1

1.1.2 HY-LandTiger

In questo paragrafo si cercherà di vedere le caratteristiche hardware di una HY-LandTiger, e come lo sviluppatore può scrivere un semplice programma in C che simuli il countdown di un contatore, mettendo in risalto l'ambiente di sviluppo più idoneo per tale tecnologia. La Fig.2 (foto ufficiale HAOYU) mostra il modello LandTiger NXP LP1768 V2.0.

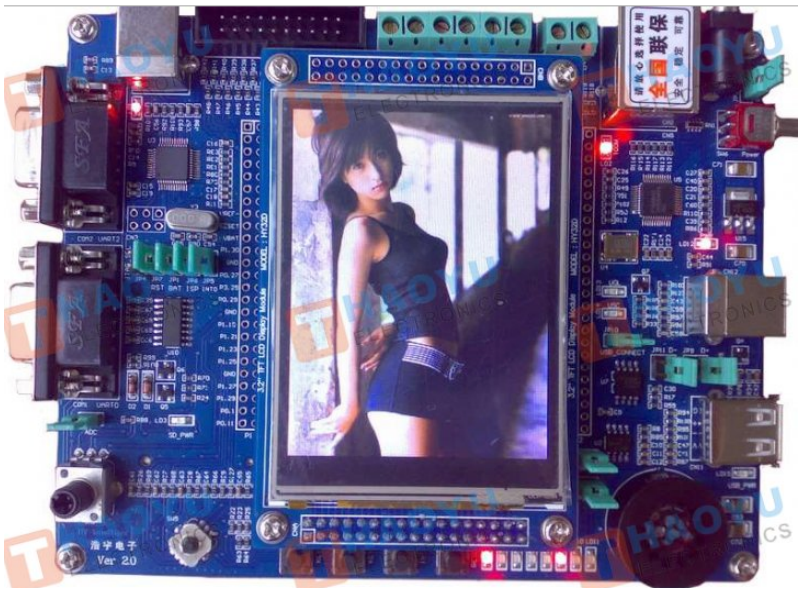


Fig.2

Il cuore della scheda è basato su un processore ARM della NXP modello LP1768 RISC a 32bit 100MHz. Sovente si commette l'errore di pensare ad ARM (Advanced RISC Machine) come ad una marca di processori, mentre invece ARM è un processore progettato dalla ARM Holdings, mentre la sua realizzazione è affidata a molti vendor come NXP, sotto marca Philips, Samsung, STMicroelectronics e tantissime altre aziende che hanno dato vita ad una ARM Partnership. La famiglia dei processori ARM è presente in quasi tutti i device mobili e si è ben presto sviluppata fino ai 64bit multi core, pietra cardine dei più famosi smartphone presenti sul mercato internazionale. La scheda HY-LandTiger monta un ARM a 32bit 100MHz con architettura RISC, realizzato dalla NXP (Not Extra Pay) branch della Philips. La scheda è dotata di un display TFT 3.2" touch, due porte seriali RS232, una scheda di rete 802.3u, due porte USB 2.0, un lettore card SD/MMC, quattro pulsanti utente più un quinto di reset della scheda, otto led, una joystick utente, un potenziometro utente, un convertitore A/D e D/A, un timer, una GPIO, che permette quindi l'interfacciamento con i protocolli SPI, I^2C e UART. La memoria ram in dotazione è pari a

64KB, mentre la dimensione della flash è di 512KB, dalla quale ne consegue che l'eseguibile prodotto dal linker non può superare tale limite, ma sopratutto è impensabile l'installazione di un sistema operativo. Il trasferimento del programma dal calcolatore alla scheda avviene tramite il connettore JTAG in tempo reale sfruttando il bus Amba.
I processori ARM si dividono in due grandi famiglie:

- Application Processor
- Embedded Processor

La Fig.3 (non aggiornata) mette in relazione il progresso tecnologico delle due famiglie.

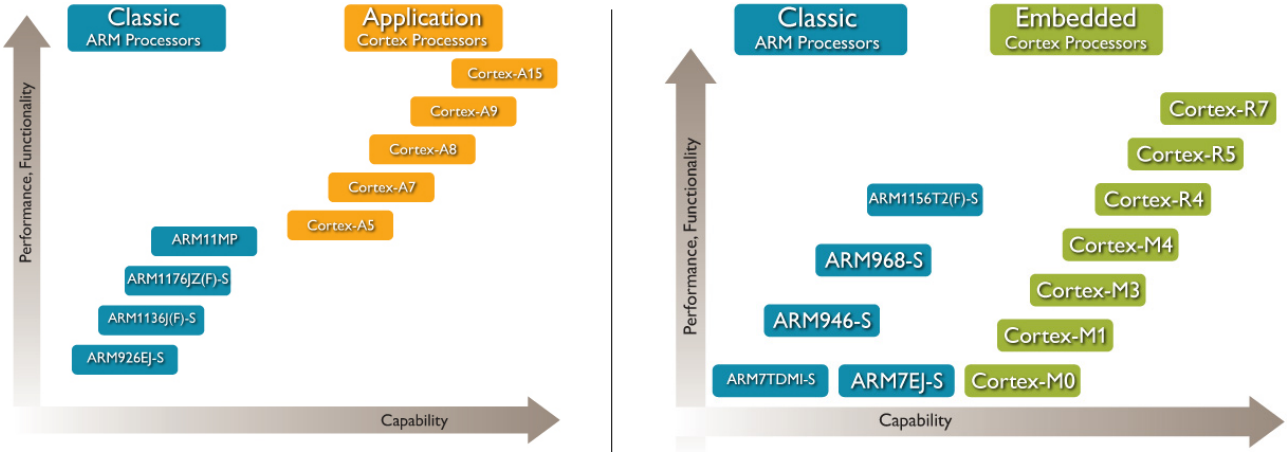


Fig.3

L'architettura di un processore ARM dipende quindi dal modello, è tutti i vendor che producono queste architetture inseriscono il processore all'interno di un System-on-Chip (SoC) assieme ad altri moduli i quali vengono tutti chiamati Embedded Core o anche IP (Intellectual Property). La Fig.4 mostra un esempio di SoC, mentre dal layout di Fig.5 si evince in modo chiaro il ruolo del SoC, con la presenza del processore, delle memorie, dei convertitori A/D e D/A e di altri eventuali moduli IP. In generale quindi possiamo dire che sulla base delle caratteristiche della memoria flash, è possibile o meno valutare l'installazione di un sistema operativo, cosa che con la scheda HY-LandTiger è impossibile, visto che con mezza mega byte si riesce a fare poche cose. E' interessante lo schema di Fig.6, il quale mette in relazione le componenti fondamentali di un generico SoC che utilizza un processore ARM Cortex-3 con la memoria ram e la flash.

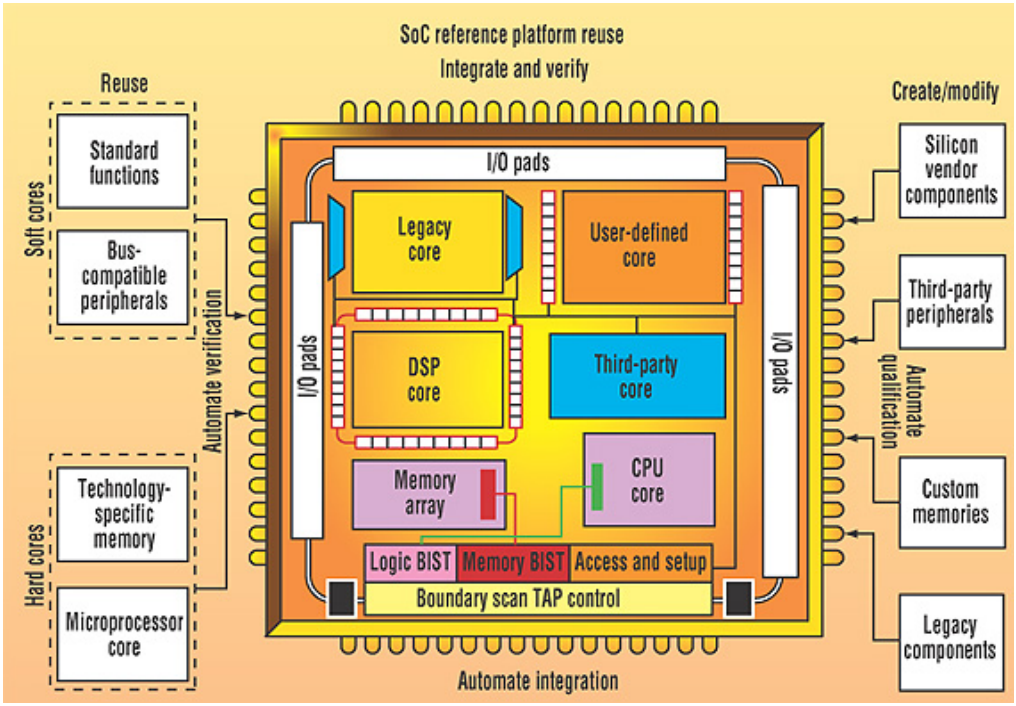

SoC reference platform reuse
Integrate and verify
Reuse
Create/modify
Soft cores
Standard functions
Bus-compatible peripherals
Hard cores
Technology-specific memory
Microprocessor core
Automate verification
I/O pads
Legacy core
User-defined core
DSP core
Third-party core
Memory array
CPU core
Logic BIST
Memory BIST
Access and setup
Boundary scan TAP control
Silicon vendor components
Third-party peripherals
Custom memories
Legacy components
Automate qualification
Automate integration

Fig.4

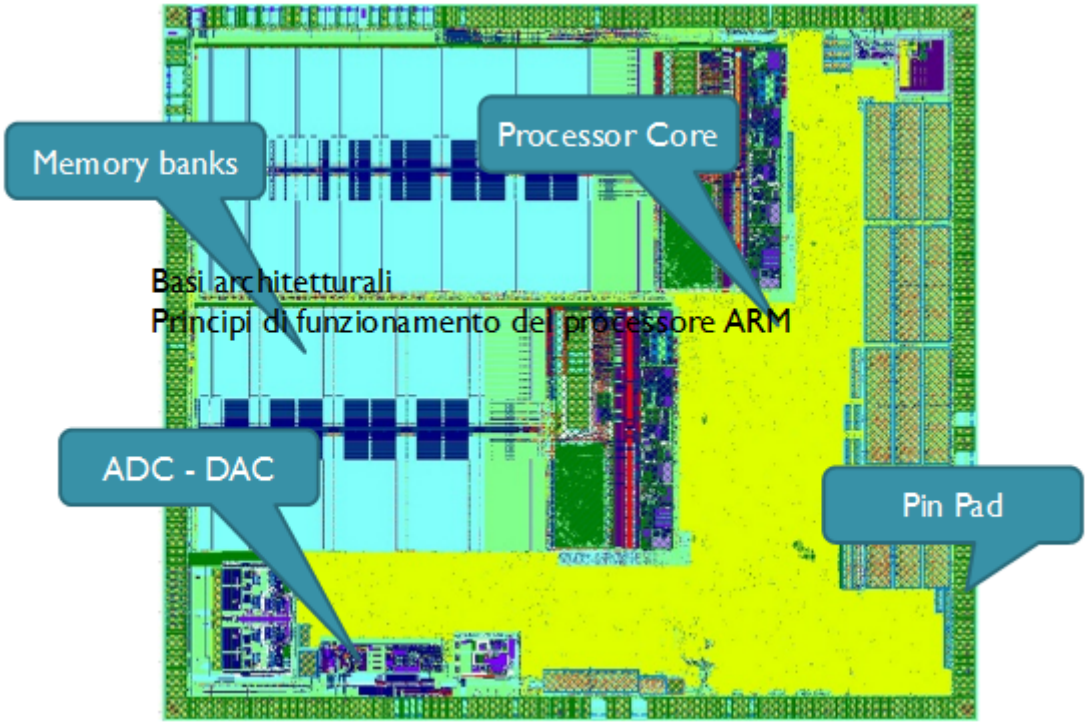

Memory banks
Processor Core
Basi architetturali
Principi di funzionamento del processore ARM
ADC - DAC
Pin Pad

Fig.5

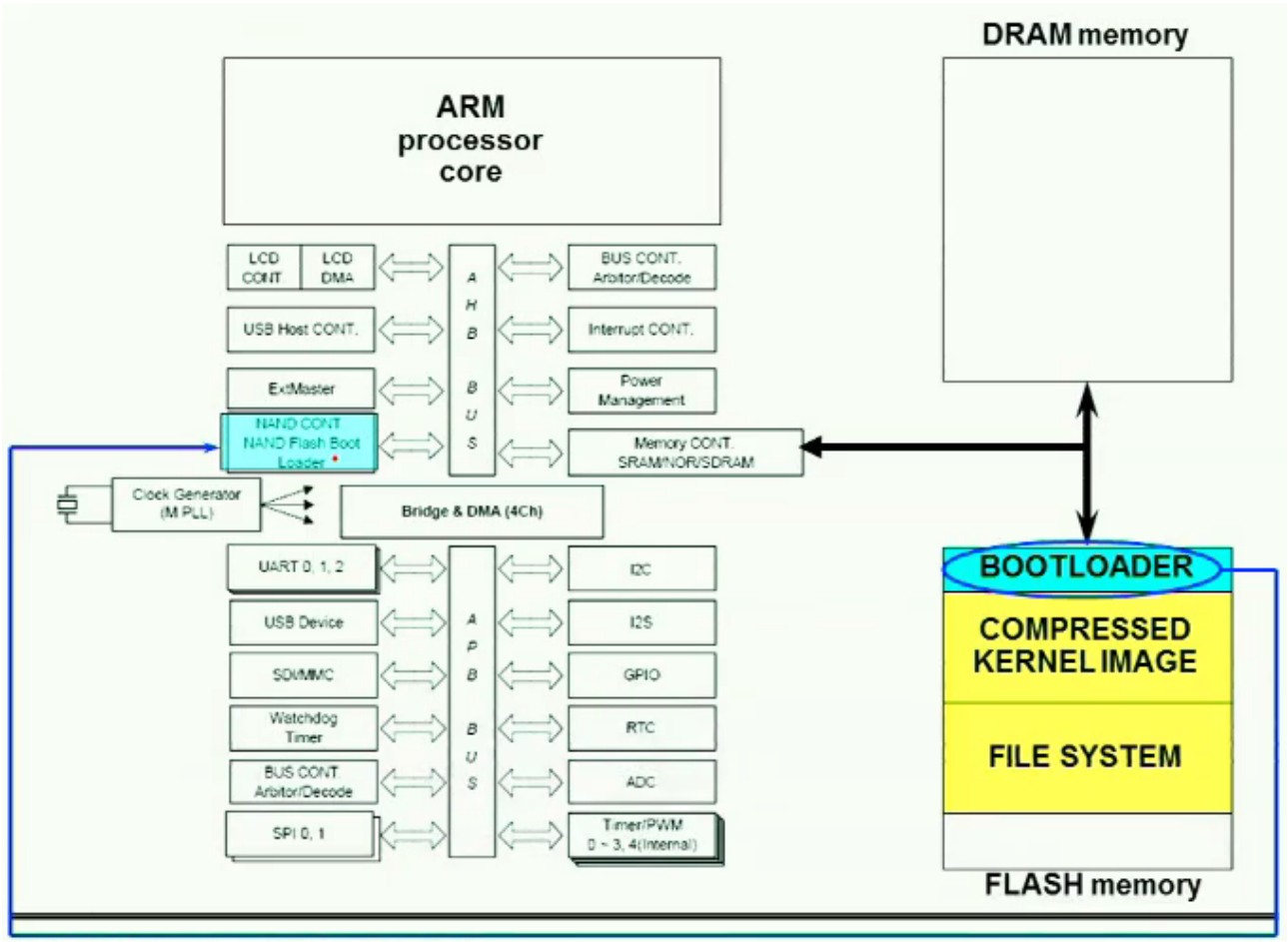


Fig.6

Il processore ARM presente nel SoC utilizza due tipi di bus Amba (AHB), il primo ad alta velocità, mentre il secondo per il collegamento con l'esterno, quindi con velocità ridotta. I due bus sono separati da un modulo Bridge. La dimensione della flash condiziona la presenza o meno del sistema operativo, in ogni caso è presente un modulo di boot, chiamato bootloader, che carica in un modulo IP del SoC del codice col fine di inizializzare la piattaforma base, in termini di configurazione del controllore della memoria, configurazione degli interrupt, configurazione dei bus Amba, ma cosa più importante avvia il processore. La Fig.7 riassume questi concetti. La CPU ARM, avendo a disposizione i periferici pronti per funzionare, procede, se presente un sistema operativo, alla decompressione dell'immagine del kernel e al successivo caricamento in memoria ram. Nel caso del HY-LandTiger, il processore carica l'immagine del processo in memoria. Si veda la Fig.8 che riassume quest'ultimo passaggio.

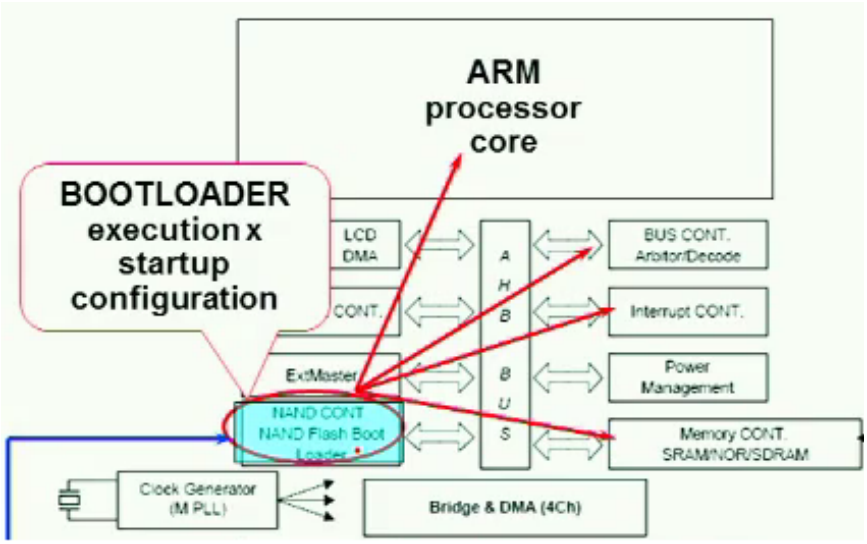

ARM
processor
core
BOOTLOADER
execution x
startup
configuration
LCD
DMA
CONT.
ExtMaster
NAND CONT
NAND Flash Boot
Loader
A
H
B
B
U
S
BUS CONT.
Arbitor/Decode
Interrupt CONT.
Power
Management
Memory CONT.
SRAM/NOR/SDRAM
Clock Generator
(M PLL)
Bridge & DMA (4Ch)

Fig.7

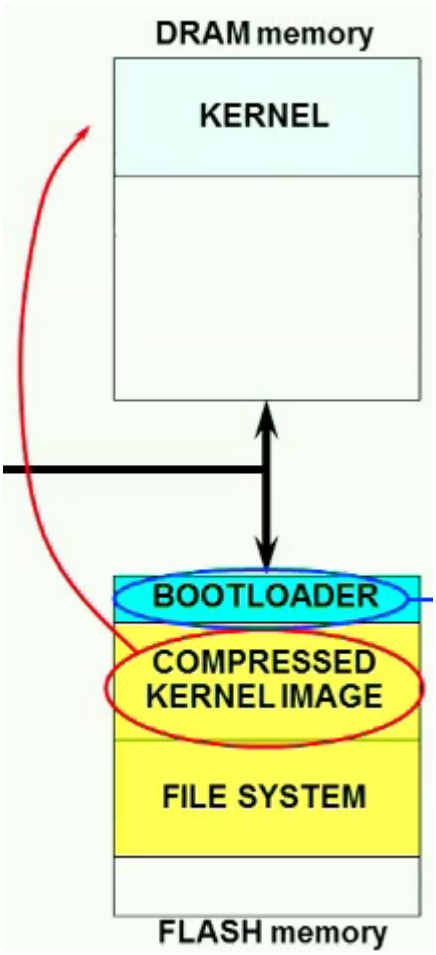

DRAM memory
KERNEL
BOOTLOADER
COMPRESSED
KERNEL IMAGE
FILE SYSTEM
FLASH memory

Fig.8

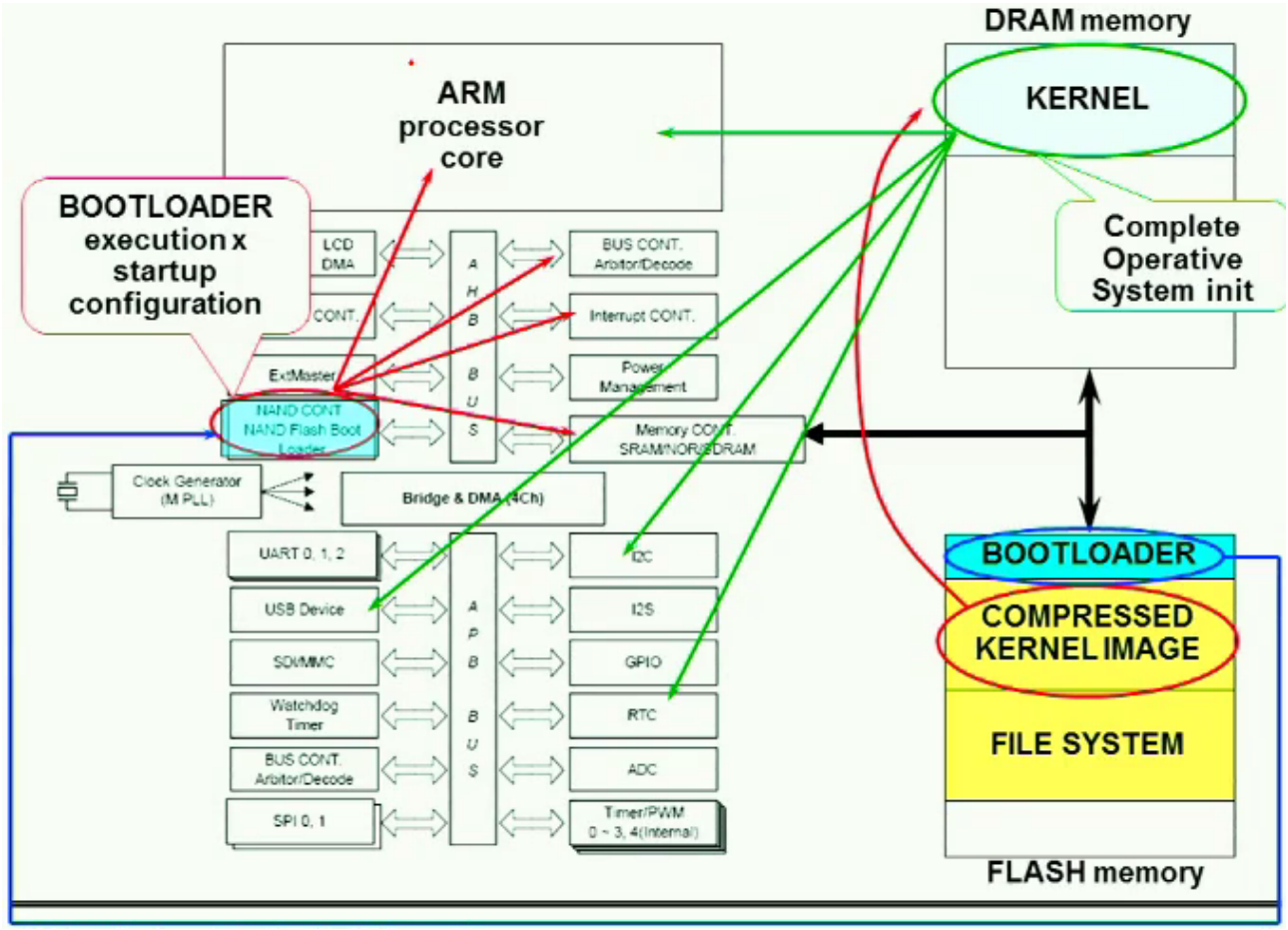


Fig.9

In Fig.9 una volta che il kernel è operativo, la piattaforma viene completamente inizializzata, avviando le routine dei dispositivi principali, quali USB, RTC, I^2C e l'impostazione dei registri base e del livello di privilegio nel processore ARM.

Fino a questo momento è stato dato per scontato che sistema operativo, da adesso O.S., o processo fossero già presenti nella memoria flash, quindi è scontato chiedersi in che modo lo sviluppatore riesce a trasferire il processo o caricare il sistema operativo dal calcolatore verso la scheda HY-LandTiger. La risposta arriva direttamente dalla Fig.10 che mostra la presenza di un connettore ad hoc, chiamato JTAG, al quale è connesso un Boundary Scan il quale serialmente permette di caricare le informazioni nella flash memory. Lo standard che definisce il Boundary Scan è il IEEE 1149.1, anche se su alcuni SoC è presente un altro standard identificato dalla sigla IEEE 1150 SECT. Il connettore JTAG è probabile che sia di tipo USB così da agevolare la connessione verso il personal computer. In Fig.11 è mostrata la locazione delle varie componenti, step iniziale per qualsiasi tipo di interfacciamento verso la scheda embedded. Essendo il processore ARM

presente anche sul Raspberry Pi 3, l'esposizione di questo paragrafo è un requisito a cui il lettore deve porre attenzione, anche se la trattazione sugli ARM richiederebbe centinaia e centinaia di pagine e quindi un manuale specifico per il modello del processore.

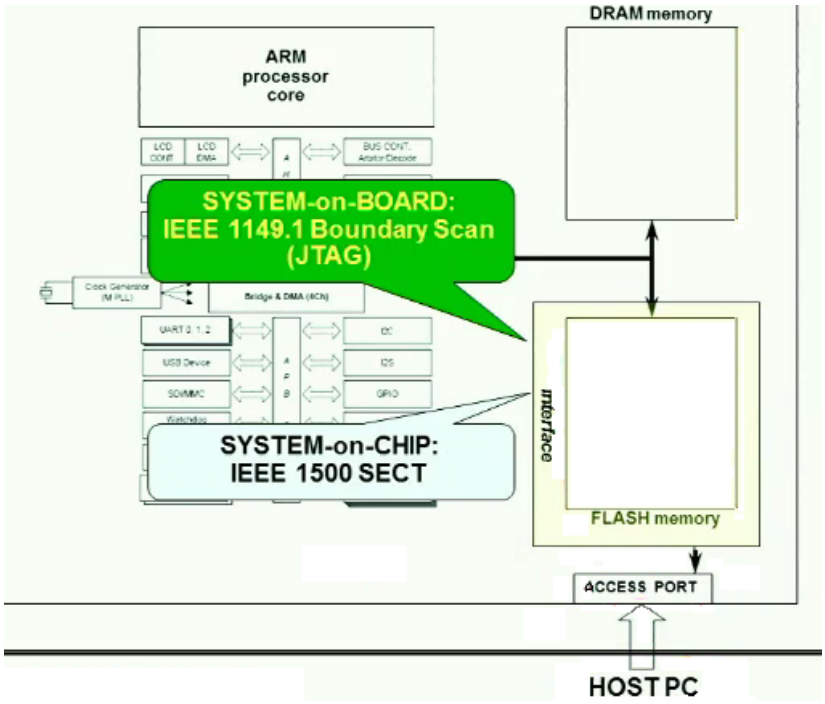


Fig.10

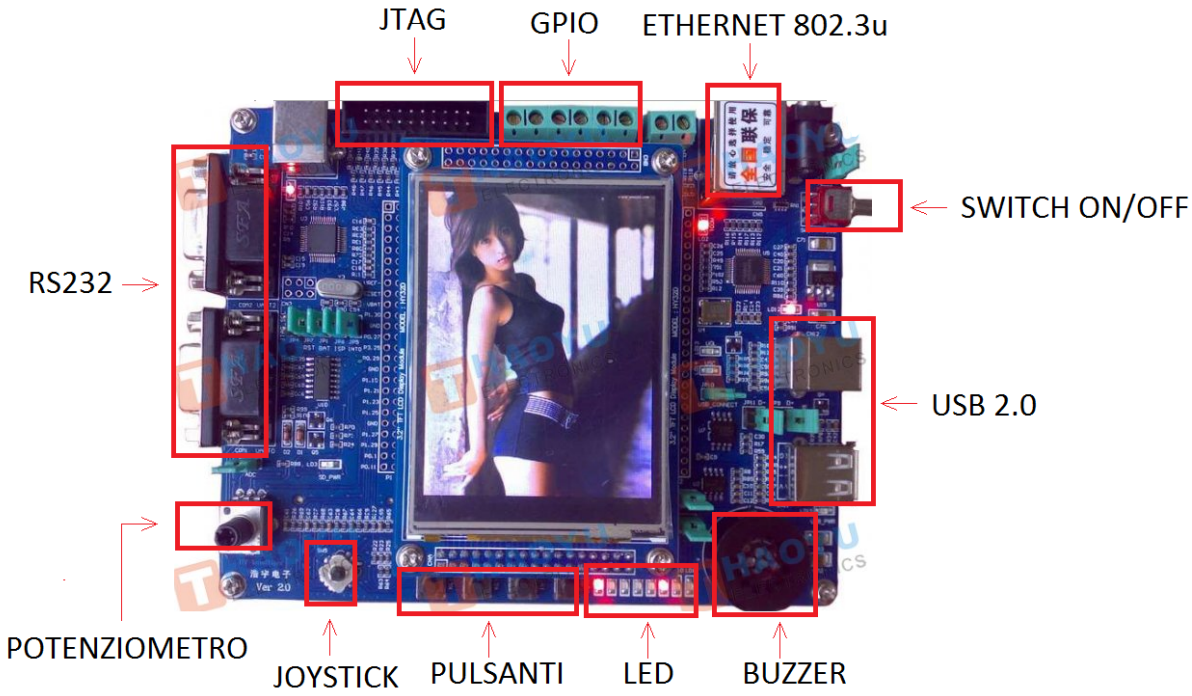


Fig.11

1.1.3 Programmazione in C con µVision4

Lo scopo di questo paragrafo è mostrare come è possibile realizzare un semplice programma in C, tramite un apposito ambiente di sviluppo, e trasferire l'eseguibile sulla scheda HY-LandTiger. Il software utilizzato in questa trattazione è la versione free di Keil µVision 4.74, anche se sul sito ufficiale http://www2.keil.com/mdk5/uvision/ è possibile scaricare la versione 5. La release free funziona completamente con l'unico vincolo sulla dimensione dell'eseguibile che non può superare i 16KB, limite che per un progetto di media grandezza e con molte librerie può essere subito raggiunto. Il linker del compilatore C di questo IDE, effettua il collegamento statico delle librerie, le quali vengono quindi conglobate all'interno dell'eseguibile facendolo aumentare di dimensione, con il vantaggio di un avvio del processo molto veloce, anche se il processo è l'unico codice in esecuzione sulla scheda essendo questa sprovvista di sistema operativo. Nei sistemi embedded con O.S., è possibile decidere di creare codice che implementi il collegamento statico, ma anche quello dinamico, il quale permette al Loader del processo di caricare on the fly, la libreria necessaria o, nel caso di processi plug-in, di caricare a tempo di esecuzione la libreria dinamica.

La Fig.12 mostra la realizzazione di un progetto "Countdown" nel quale un semplice contatore viene decrementato da 20 fino a 0 per poi stampare sul display un messaggio.

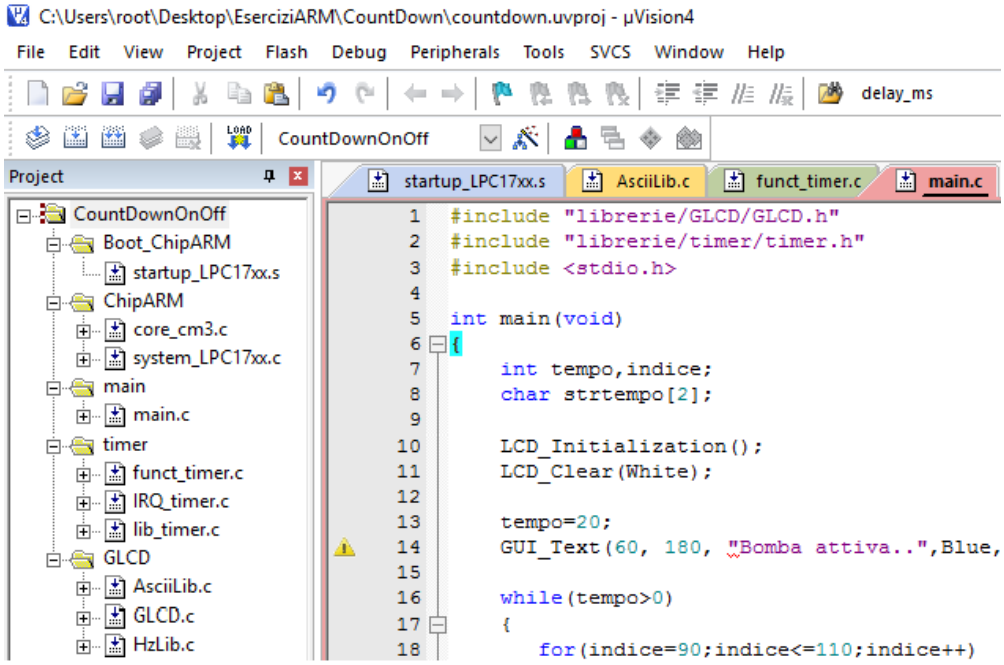


Fig.12

Si noti che nella sezione "Project" si creano una serie di sotto directory "Boot_ChipARM/ChipARM/timer/GLCD" che contengono i sorgenti per utilizzare le periferiche, partendo dall'inizializzazione della memoria, dalla configurazione della piattaforma in base al chip, nel caso specifico un Cortex-M3 LP1768, fino alla gestione del display. Questi sorgenti in linguaggio C vengono forniti dal produttore della scheda. La directory "main" contiene invece il modulo "main.c" il quale implementa la funzione "main()" dentro la quale viene inserito, in modo semplicistico, il codice dimostrativo dell'esempio "Countdown". Il codice C sottostante sfrutta delle funzioni preparate dal programmatore per accedere in modo semplice al timer di sistema e al display.

```
#include "librerie/GLCD/GLCD.h"
#include "librerie/timer/timer.h"
#include <stdio.h>

int main(void)
{
    int tempo,indice;
    char strtempo[2];

    LCD_Initialization();
    LCD_Clear(White);

    tempo=20;
    GUI_Text(60, 180, "Bomba attiva..",Blue, White);

    while(tempo>0)
    {
       for(indice=90;indice<=110;indice++)
         GUI_Text(indice, 220, " ",Blue, White);

       sprintf(strtempo, "%i", tempo);
       GUI_Text(100, 220, (uint8_t*)strtempo,Blue, White);

       delayMs(0, 1000);

       tempo--;
    }

    LCD_Clear(White);
    GUI_Text(80, 180, "Boom!!!",Blue, White);
    GUI_Text(60, 200, "You are died..",Blue, White);

    return 1;
}
```

La funzione "LCD_Inizialization()" e "LCD_Clear(White)" impostano rispettivamente la risoluzione del display con sfrondo bianco. La funzione "GUI_Text()" serve per scrivere del testo alla coordinate in pixel X,Y. La funzione "delayMs" imposta un ritardo di 1000

millisecondi sul timer numero 0. La logica del programma è banale, si cicla finché il contatore "tempo", impostato a 20, non viene decrementato a 0, il tutto ritardando di un secondo così da dare una corretta percezione all'utente. Il valore del contatore, prima di venire stampato a video, viene convertito da "int" a "char[]" tramite la funzione del C "sprintf" visto che il prototipo della "GUI_Text()" richiede un "char[]" e non una variabile intera. Il ciclo iterativo "for" serve per pulire la decina, così che non sia visibile quando il contatore avrà un valore inferiore a 10.
Terminata la scrittura del programma si procede alla compilazione e linking del codice tramite il pulsante "Build" di Fig.13.

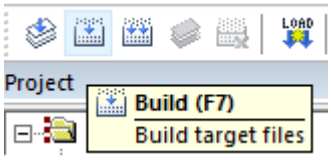


Fig.13

Il risultato del linking è la produzione di un file con estensione ".axf" che dovrà venire caricato nella flash memory della scheda. Si veda la Fig.14.

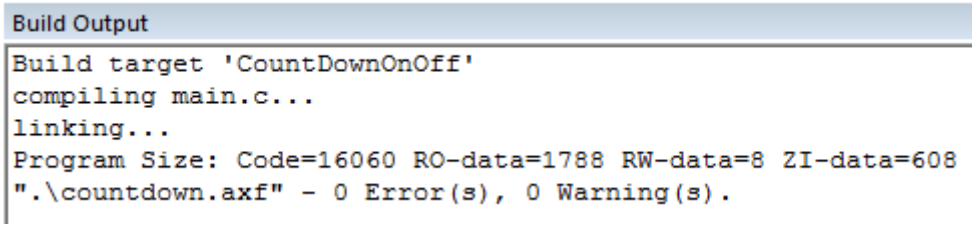


Fig.14

Il passo successivo è effettuare il debug del programma in tempo reale, senza utilizzare la parte di simulazione presente all'interno dell'ambiente Keil µVision. Per fare questo è necessario dotarsi di un debugger hardware, quale il RealView ULink 2 di Fig.15, ad un costo extra di 12€. Il device è facilmente reperibile sul mercato web, anche se è capitato che prendendone un lotto cospicuo, la piatta flat fosse invertita a causa di un difetto di assemblaggio, con la conseguenza che l'ambiente IDE non riconoscesse dispositivo. Il sistema operativo Windows riconosce il debugger hardware come device USB senza richiedere nessun tipo di driver.

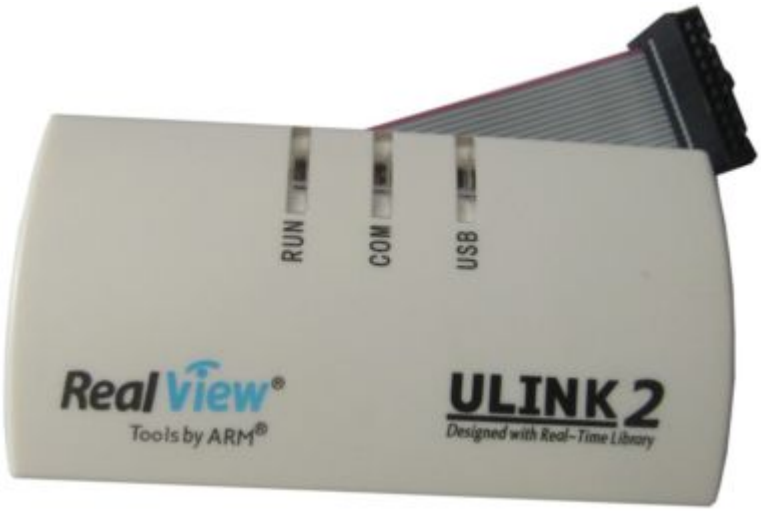


Fig.15

Dal menù "Flash" dell'IDE, selezionare la voce "Configure Flash Tools..." come da Fig.16.

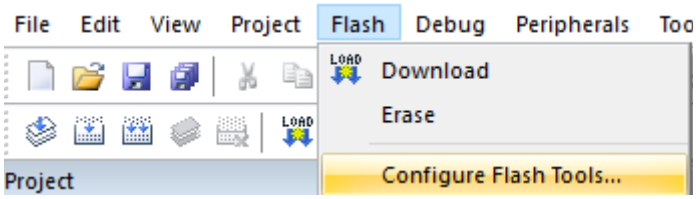


Fig.16

Nella finestra di configurazione selezionare la voce "Debug" ed impostare come da Fig.17 l'uso del debug hardware che l'ambiente riconosce previa installazione del relativo driver da parte di Microsoft Windows.

Il passaggio finale consiste nel caricare l'eseguibile ".axf", tramite il debug hardware, direttamente nella flash memory. Per fare questo basta premere il pulsante "LOAD" di Fig.18, che produce la cancellazione della flash con il codice in essa contenuto, ed il caricamento del nuovo programma. Il tutto avviene in modo veloce come da Fig.19. E' interessante vedere il comportamento del debugger quando si preme "LOAD", infatti oltre alla spia rossa che segnala la presenza della connessione "USB", si attiva per un brevissimo lasso di tempo anche la spia verde "RUN" come si vede dalla Fig.20. Una pressione del tasto reset di Fig.21 fa si che il programma venga eseguito, come si vede nella sequenza di immagini di Fig.22. Il sistema finale composto da HY-LandTiger e debugger hardware è quello di Fig.23.

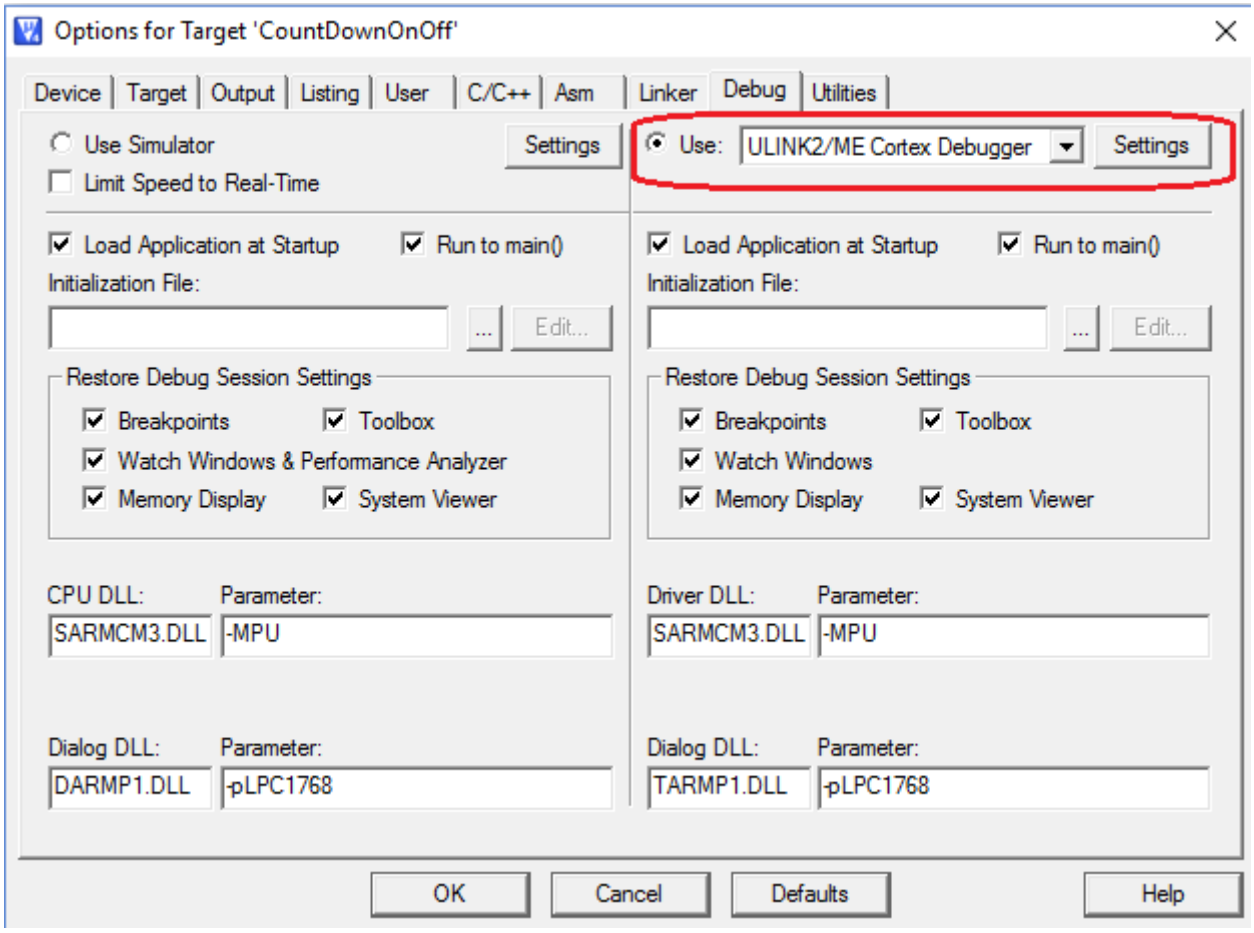


Fig.17

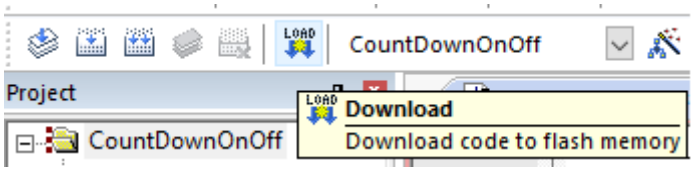


Fig.18

```
Load "C:\\Users\\root\\Desktop\\EserciziARM\\CountDown\\countdown.axf"
Erase Done.
Programming Done.
Verify OK.
```

Fig.19

Fig.20

Fig.21

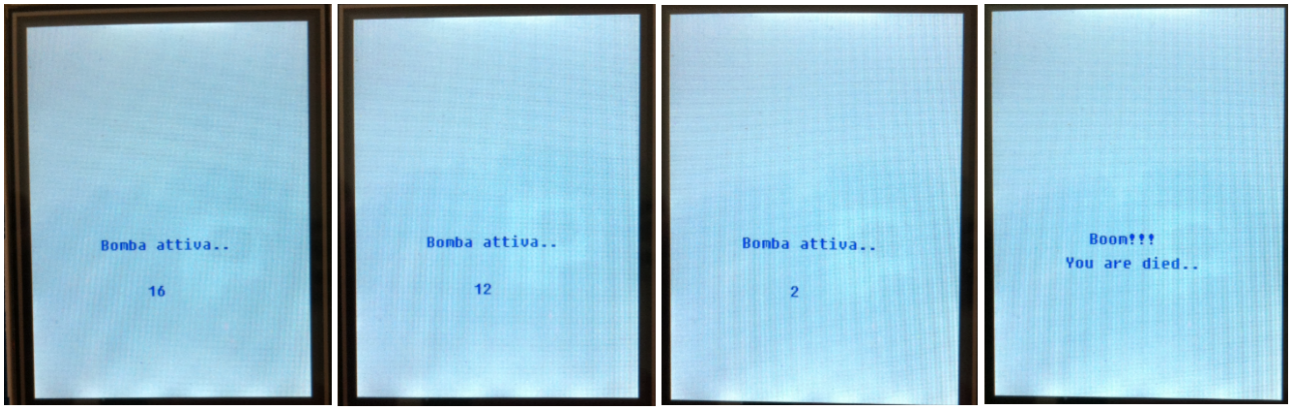


Fig.22

Fig.23

La tecnologia di questa scheda è sicuramente interessante, visto il numero elevato di periferiche a bordo e la facilità di programmazione tramite l'IDE Keil Vision. Il consumo energetico è molto basso grazie all'impiego del NXP ARM Cortex-M3, ed il costo è abbordabile se paragonato con sistemi che poi richiedono pulsanti, led o convertitori A/D-D/A.

L'unico neo è la scarsa documentazione reperibile in rete che rende questo prodotto uno scoglio non proprio facile da superare per coloro che sono alle prime armi nel mondo della programmazione dei device embedded. Il tutto accompagnato dall'uso del solo linguaggio C e ARM Assembly.

1.2 Raspberry Pi e periferici

Lo scopo di questo paragrafo è introdurre il sistema embedded Raspberry Pi descrivendo in modo sintetico le differenze tra le varie versioni, cercando di evidenziare le motivazioni che lo rendono più idoneo rispetto al HY-LandTiger. Raspberry Pi è stato progettato dall'omonima fondazione in collaborazione con l'università di Cambrdige col fine di offrire uno strumento, economicamente sostenibile, per l'apprendimento dell'informatica nelle scuole. La Fig.24 mostra le dimensioni di un Pi 3 modello B, dal quale si vede chiaramente che l'intera schedina è grande come una carta di credito.

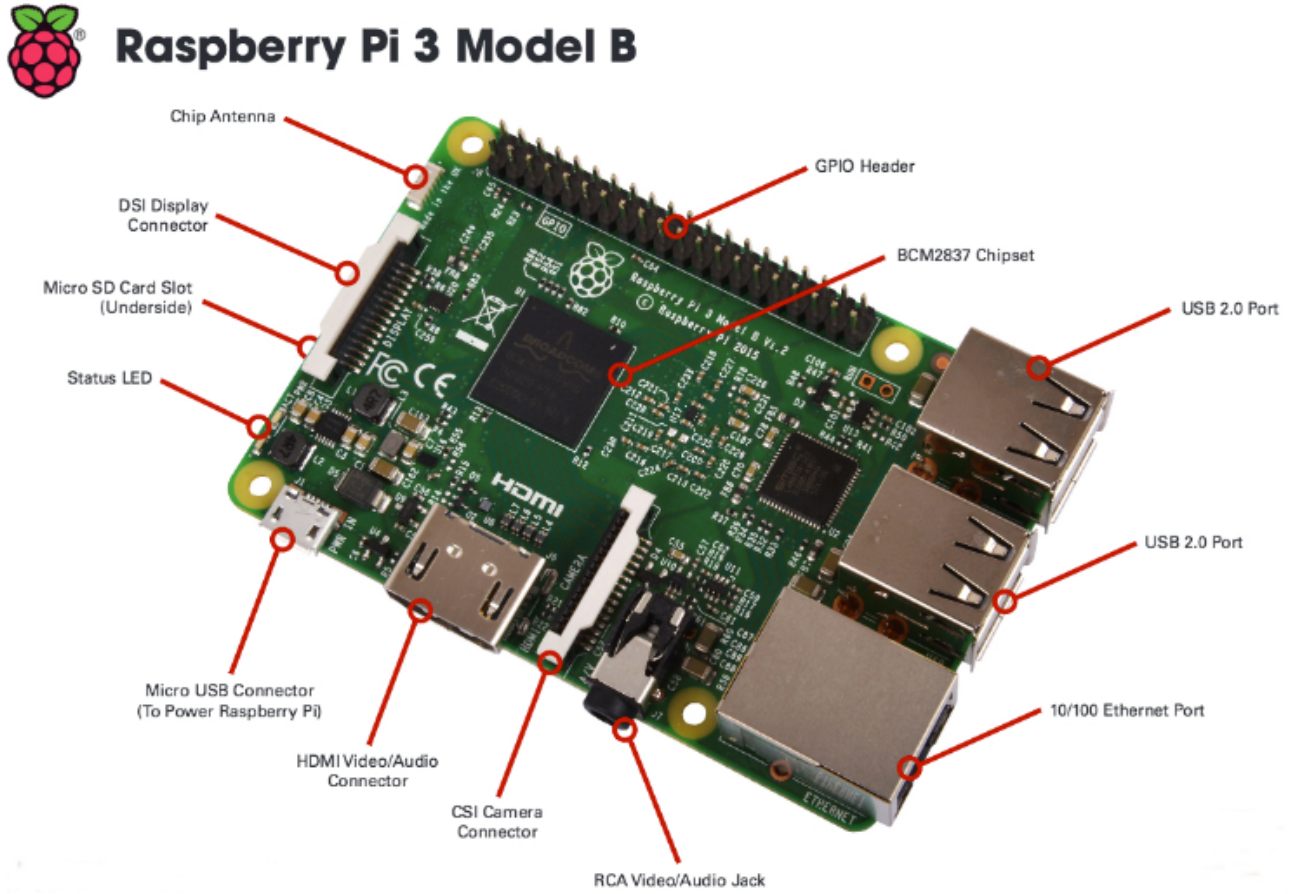


Fig.24

Il cuore di ogni Pi si basa su un SoC Broadcom che monta all'interno un ARM con caratteristiche che variano da modello a modello. Nella Tab.2 sono riportate le specifiche tecniche più interessanti tra i modelli che sono stati prodotti in questi ultimi anni. Il costo del device è indicativo.

	Pi 0	Pi 1 modello B+	Pi 2 modello B	Pi 3 modello B
SoC	Broadcom BCM2835	Broadcom BCM2835	Broadcom BCM2836	Broadcom BCM2837
CPU	ARM1176JZF-S single core 32bit 1GHz	ARM1176JZF-S single core 32bit 700MHz	ARM Cortex-A7 32bit quad-core 900MHz	ARM Cortex-A53 64bit quad-core 1.2GHz
SDRAM	512MB condivisa con GPU	512MB condivisa con GPU	1GB condivisa con GPU	1GB condivisa con GPU
USB 2.0	1 micro-USB	4 porte tramite hub integrato	4 porte tramite hub integrato	4 porte tramite hub integrato
Ethernet 802.3u	assente	presente	presente	presente
Wireless 802.11n	assente	assente	assente	presente
Bluetooth 4.1	assente	assente	assente	presente
Ingresso audio	tramite bus I^2S	tramite bus I^2S	tramite bus I^2S	tramite bus I^2S
Ingresso video	assente	presente	presente	presente
Uscita video	mini-HDMI	HDMI	HDMI	HDMI
Uscita audio	mini-HDMI	tramite bus I^2S	tramite bus I^2S	tramite bus I^2S
Memoria	micro-SD	micro-SD	micro-SD	micro-SD
GPIO	GPIO a 40pin	GPIO a 40pin	GPIO a 40pin	GPIO a 40pin
Assorbimento	~160ma (0.8W)	~600ma (3W)	~800ma (4W)	~800ma (4W)
Sistema operativo	Linux/Windows 10 IoT	Linux	Linux/Windows 10 IoT	Linux/Windows 10 IoT
Costo indicativo	€ 5,00	€ 20,00	€ 35,00	€ 45,00

Tab.2

Dalla tabella si capisce che il modello Pi 3 è praticamente un computer, dotato di connessioni di rete 802.11n e 802.u, dotato di processore a 64bit con architettura quad-core, dotato di micro-SD che funge da memoria secondaria, dotato di bluetooth e con un quantitativo di ram di 1GB in condivisione con la GPU. Tutte queste caratteristiche si sposano bene per ospitare un sistema operativo quali Windows 10 IoT, versione dedicata ai sistemi embedded, che dovrà venire installata su una schedina SD di almeno 8GB. Il consumo non è esagerato, anche se paragonato con il modello 0 le differenze sono significative. L'aspetto cardine che introduce una spaccatura tra un calcolatore ed un sistema embedded è la presenza del connettore GPIO (General Purpose Input/Output) che permette di connettere il Pi ha tantissimi sensori, motori e altre schede elettroniche. E'

tutto lasciato alla fantasia del Maker, sempre però nel rispetto dei protocolli di trasmissione che saranno descritti nel seguito.

1.2.1 GPIO

In questo paragrafo è importante discutere del GPIO così che l'utente conosca il ruolo di tale connettore e come interfacciare il Pi ha vari tipi di sistemi. La conoscenza del significato dei pin del GPIO è consigliata, così come la conoscenza dei protocolli di comunicazione standard utilizzati da Raspberry e da molti altri sistemi. Avere delle buone basi di elettrotecnica può essere molto utile per un corretto uso di componenti elettronici come resistori e condensatori elettrolitici. La Fig.25 mostra i 40 pin del Pi 2 e 3.

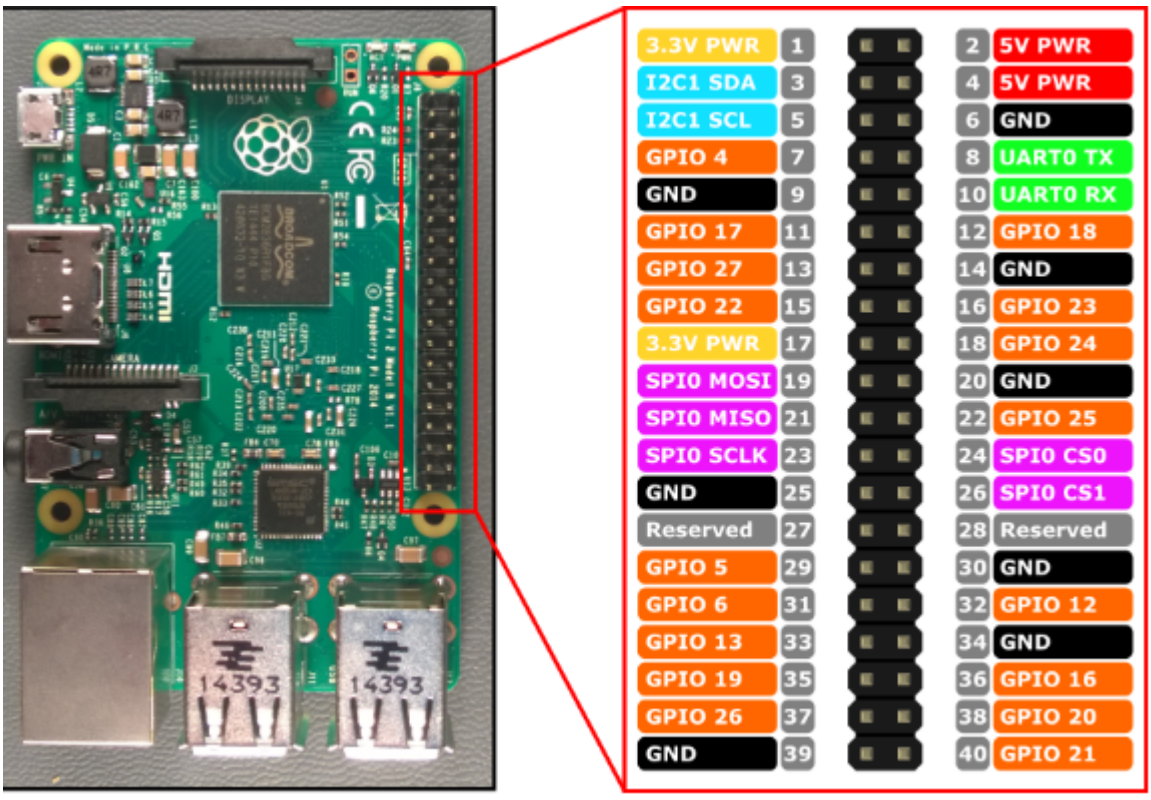


Fig.25

La logica di funzionamento è binaria, quindi lo 0 logico corrisponde a 0V, mentre 1 logico corrisponde a +3.3V e non +5V come si potrebbe erroneamente credere. Tutti i pin GPIO, ad eccezione delle masse 6,9,20,25,34, 39 e delle alimentazioni a +5V sui pin 2 e 4, funzionano a +3.3V. Attenzione a non collegare nessun dispositivo esterno che funzioni a

tensioni diverse dai +3.3V perché si può compromettere in modo irreversibile il funzionamento del pin GPIO utilizzato. In questo caso, in presenza di tensioni superiori ai +3.3V, è possibile costruire dei circuiti che possano abbassare il livello di tensione adattandolo alle specifiche del Pi. Sovente si utilizza il termine "5V not tollerant" per indicare quando esposto. La corrente massima che può erogare il Pi su pin di uscita è di 10mA, mentre sul pin di alimentazione numero 1 da +3.3V è pari a 50mA. La corrente massima sui pin 2 e 4 da +5V è quella che arriva dall'alimentazione, anche se parte di questa viene utilizzata per alimentare il processore, con un dispendio quindi di circa 800mA per il Pi 3. Diviene quindi fondamentale il tipo di alimentatore che utilizzate per il Raspberry, il modello classico fornisce 1A di corrente, anche se potrebbe essere più vantaggioso optare per un'alimentatore da 2.5A ad un costo leggermente maggiore. In ogni caso un'alimentazione da 1A, si potrà utilizzare circa 200mA per i device che l'utente connette alle porte USB, valore che può rilevarsi molto basso in presenza di più dispositivi USB, circostanza che potrebbe poi obbligare l'acquisto di un USB Hub auto alimentato. Parte di questi problemi può quindi venire risolto proprio acquistando un'alimentatore da 2.5A e non il classico da 1A che non è in dotazione all'acquisto del Pi, ma che spesso è offerto in kit su molti siti specializzati di elettronica. La Fig.26 è un classico esempio di alimentatore USA per Pi 3 da 2.5A con presi micro-USB con un costo di circa € 6,00 sul mercato cinese, al quale va aggiunto ovviamente l'adattatore di presa italiano per circa € 2,00.

Fig.26

Scelto l'alimentatore non resta che collegarlo come fatto in Fig.27.

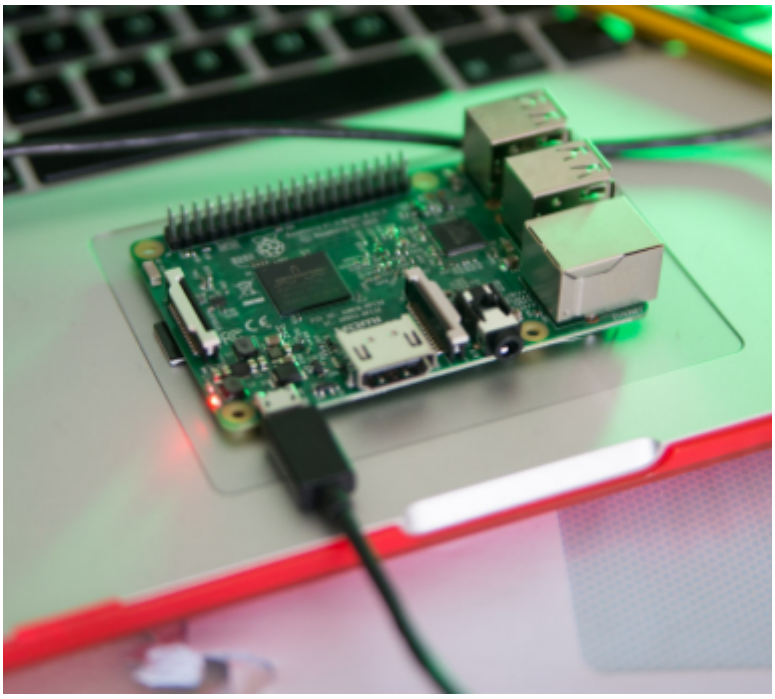


Fig.27

1.2.2 I^2C, SPI e UART

In questo paragrafo è importante menzionare i tre protocolli di comunicazione utilizzati dal Pi. Dalla Fig.25 relativa al GPIO, si vedono colorati in modo diverso tre gruppi di pin, ad identificare i tre protocolli seriali I^2C (Inter Integrated Circuit), SPI (Serial Peripheral Interface) e UART (Universal Asynchronous Receiver Trasmitter). Un programma che accenda un led esterno è banalmente realizzabile inviando lo stato logico alto su un pin d'uscita della GPIO a cui ovviamente va collegato in led con in serie un resistore. Verrà realizzato successivamente questo primo circuito in C# come testing della scheda. Quello che è importante sottolineare, è che una simile configurazione non necessità di una serie di bit in uscita e quindi una serie di stati logici diversi, ma un semplice livello logico alto. L'interfacciamento con dispositivi esterni complessi non può realizzarsi con un semplice livello logico, ma ha bisogno di uno stream di livello logici, ossia uno stream di byte seriali, ecco quindi la necessità di utilizzare un protocollo seriale di comunicazione. La transazione da uno stato logico all'altro avviene in sincronismo con un clock di sistema.

Questa logica seriale viene utilizzata oggi giorno per trasferire in burst molte quantità di dati, come capita con i device USB e SATA. La prima famiglia di protocolli seriali è chiamata sincrona, con la quale si intende che assieme ai dati viene anche inviato il clock. I protocolli I^2C e SPI sono seriali sincroni. La seconda famiglia di protocolli seriali è chiamata asincrona, con la quale si intende che si decide a priori una frequenza di clock alla quale tutti i device devono sottostare, motivo per cui il clock non viene trasmesso sulla linea dati. Il protocollo UART è seriale asincrono.

I^2C è un protocollo seriale sincrono complesso con un bandwith massimo di 400Kbps, utilizzato per creare configurazioni hardware complesse nelle quali più dispositivi sono collegati contemporaneamente alle linea di trasmissione. I^2C è stato inventato dalla Philips nel 1982 e utilizza solo due linee, una dedicata al trasferimento seriale dei dati chiamata SDA (Serial Data), la seconda utilizzata dal clock SCL (Serial Clock). La Fig.25 indica in azzurro il pin numero 3 come SDA ed il pin numero 5 come SCL. Con il protocollo I^2C, un solo device funge da master, e quindi può leggere o scrivere sulla SDA, mentre tutti gli altri device fungono da slave e quindi condividono la linea. Un solo slave alla volta può utilizzare la linea SDA, gli devono restare inattivi. La Fig.28 riassume in modo semplice quanto esposto.

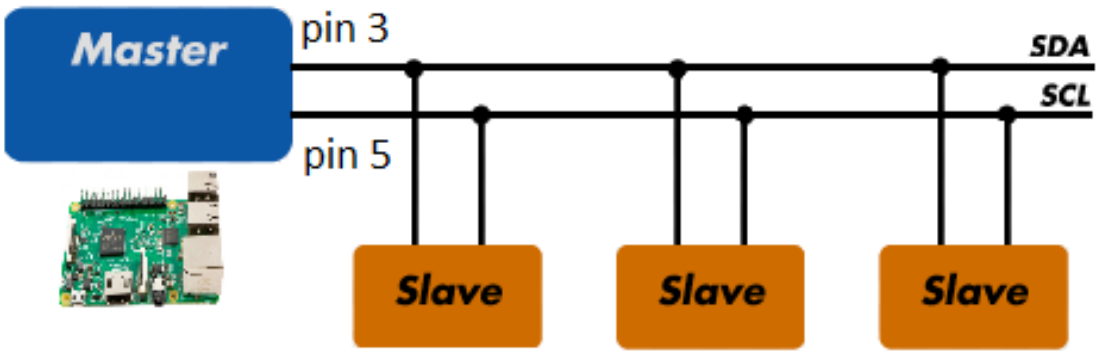


Fig.28

Il master per accedere ad un determinato slave in lettura o scrittura utilizza un determinato indirizzo reperibile nel datasheet del dispositivo esterno, il quale il più delle volte è un device a +5V che richiede quindi un corretto adattamento in tensione tramite dei chip adattatori o, più semplicemente, con un partitore di tensione che funga da voltage divider. La Fig.29 mostra un semplice esempio di questo circuito nel quale il rapporto tra i resistori

R1 e R2 deve valere 2/3 così che la tensione al capo di R2 sia pari a +3V, leggermente sotto la soglia massima dei +3.3V.

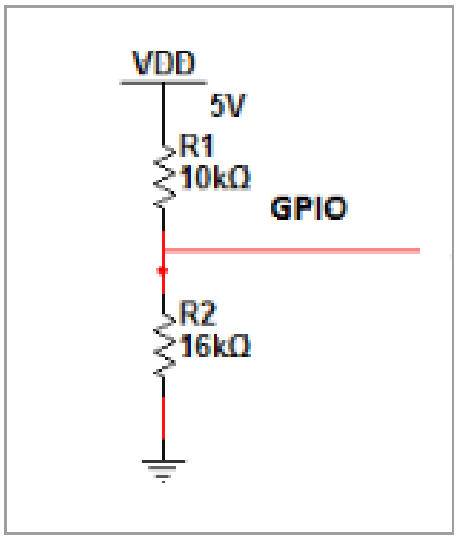


Fig.29

Per stabilire questo rapporto di 2/3 basta utilizzare le regole elementari del partitore di tensione, come riportano i calcoli sottostanti.

$$V_{R2} = V_{DD}\frac{R_2}{R_1+R_2} \Rightarrow \frac{V_{R2}}{V_{DD}} = \frac{R_2}{R_1+R_2} \Rightarrow \frac{V_{DD}}{V_{R2}} = \frac{R_1+R_2}{R_2}$$

$$\frac{V_{DD}}{V_{R2}} = \frac{R_1}{R_2} + 1 = \frac{5_V}{3_V} \Rightarrow \frac{R_1}{R_2} = \frac{5_V}{3_V} - 1 = 0.66 \Rightarrow \frac{R_1}{R_2} = 0.66 = \frac{2}{3}$$

$$\frac{R_1}{R_2} = \frac{2}{3} \Rightarrow \boxed{R_1 = \tfrac{2}{3}R_2}$$

Sulla base del circuito di Fig.29 il rapporto tra i resistori commerciali 10KΩ e 16KΩ vale circa 2/3.

L'adattamento in tensione è il primo dei problemi che l'utente deve considerare, onde evitare danneggiare seriamente la GPIO del Pi. Un secondo problema è la compatibilità elettrica in presenza di alta impedenza, con la necessità di inserire una resistenza R_{PU} di pull-up. Per capire in modo chiaro questa problematica, è bene ricordare che il grande vantaggio del I^2C è la possibilità di collegare fino a 127 periferiche allo stesso bus dati

SDA e allo stesso bus di clock SCL. Il Raspberry funge da master ed è in grado di dialogare con un solo device al volta, mentre i restanti devono obbligatoriamente restare inattivi, che da un punto di vista elettrico significa che devono restare in alta impedenza così da non interferire sulla trasmissione in essere tra master e slave attivo. Procedendo con ordine la Fig.30 evidenzia la problematica in questione, dove sul bus dati SDA sono stati collegati, a titolo di esempio, tre device generici indicati con le lettere A, B e C. Quando un device è attivo, significa che elettricamente è connesso al bus, mentre dalla figura sottostante si evince che tutti i dispositivi sono fisicamente sconnessi tramite la presenza di un deviatore logico. Questa configurazione hardware comporta che lo stato logico complessivo nel circuito digitale sia pari ad alta impedenza, stato indicato in molti testi con la lettera Z.

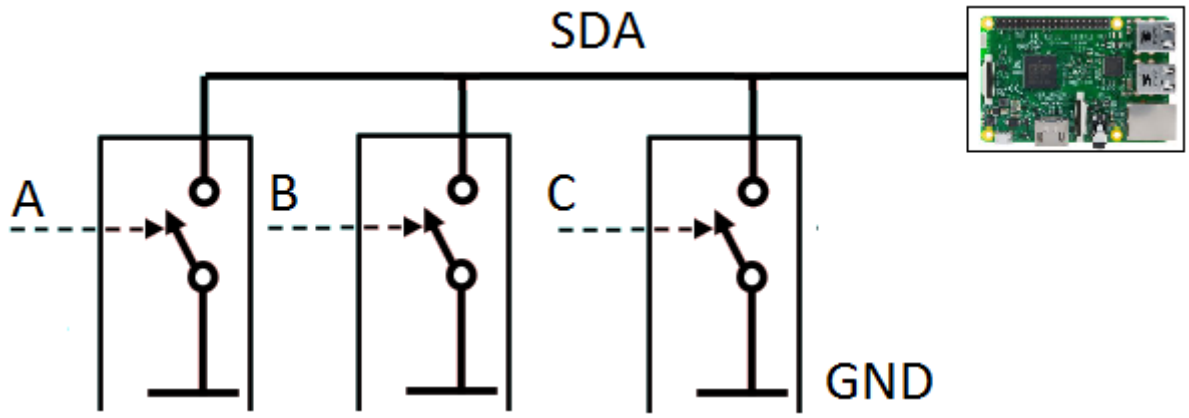


Fig.30

Un circuito digitale non è logicamente compatibile con lo stato di alta impedenza, visto che il master avrà la necessità di lavorare con il solo stato alto (H) o stato basso (L). Per risolvere questa problematica dell'inattività di tutti gli slave con conseguente alta impedenza, è necessario inserire un resistore chiamato di pull-up il quale permette in caso di inattività di tutte le periferiche di avere uno stato alto grazie alla caduta di tensione che si genera ai capi della resistenza di pull-up R_{PU}. La Fig.31 mostra la modifica da apportare al circuito perché sia logicamente compatibile. Basta un solo resistore R_{PU} connesso al bus per fare in modo che $V_O=V_{AL}$ così da avere uno stato logico alto (H) quando tutti i device slave sono inattivi. Nel momento in cui almeno uno slave diviene attivo, lo stato logico diviene basso (L) visto che il deviatore messo in corto circuito "vince" rispetto alla resistenza di pull-up portando $V_O=0V$. Il fatto che si è utilizzato il termine "almeno uno" è perché da un punto di vista logico, anche se vi fossero altri slave attivi, il risultato non

cambierebbe, ossia si avrebbe sempre uno stato logico basso (L) con V_O=0V. Ovviamente da un punto di vista funzionale del sistema, l'uso in lettura o scrittura della linea da parte di più dispositivi è vivamente sconsigliato.

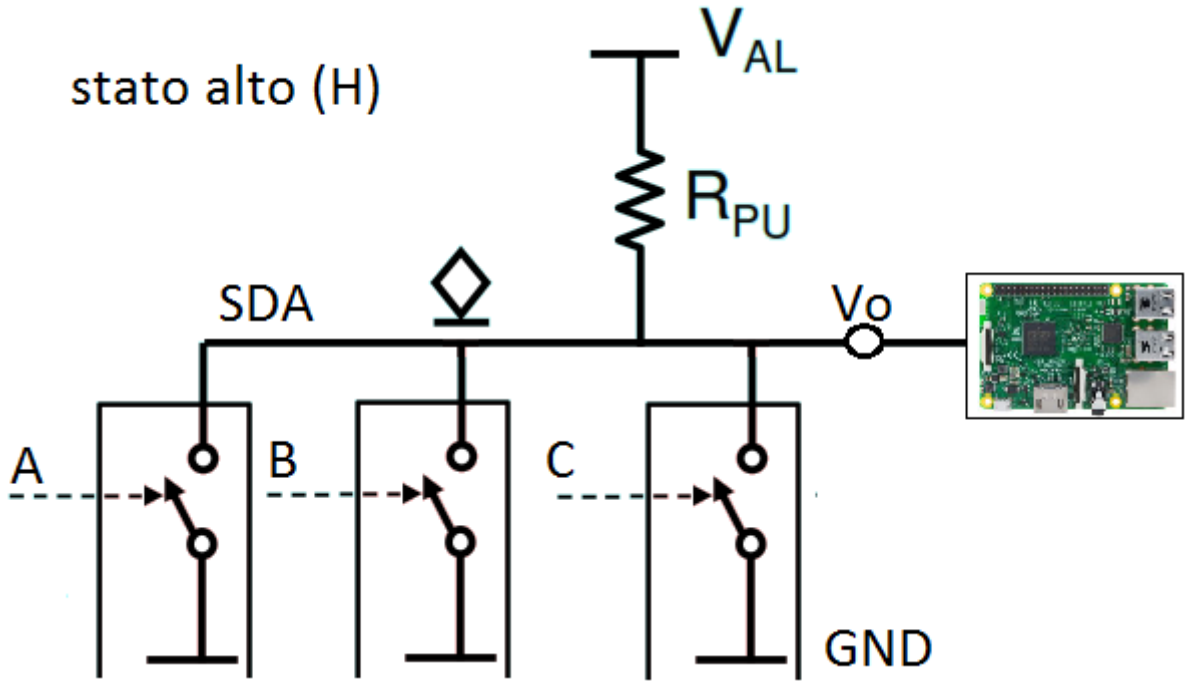


Fig.31

La Fig.32 mostra il comportamento dello stato logico basso quando il device A diviene attivo, portando a 0V la tensione V_O.

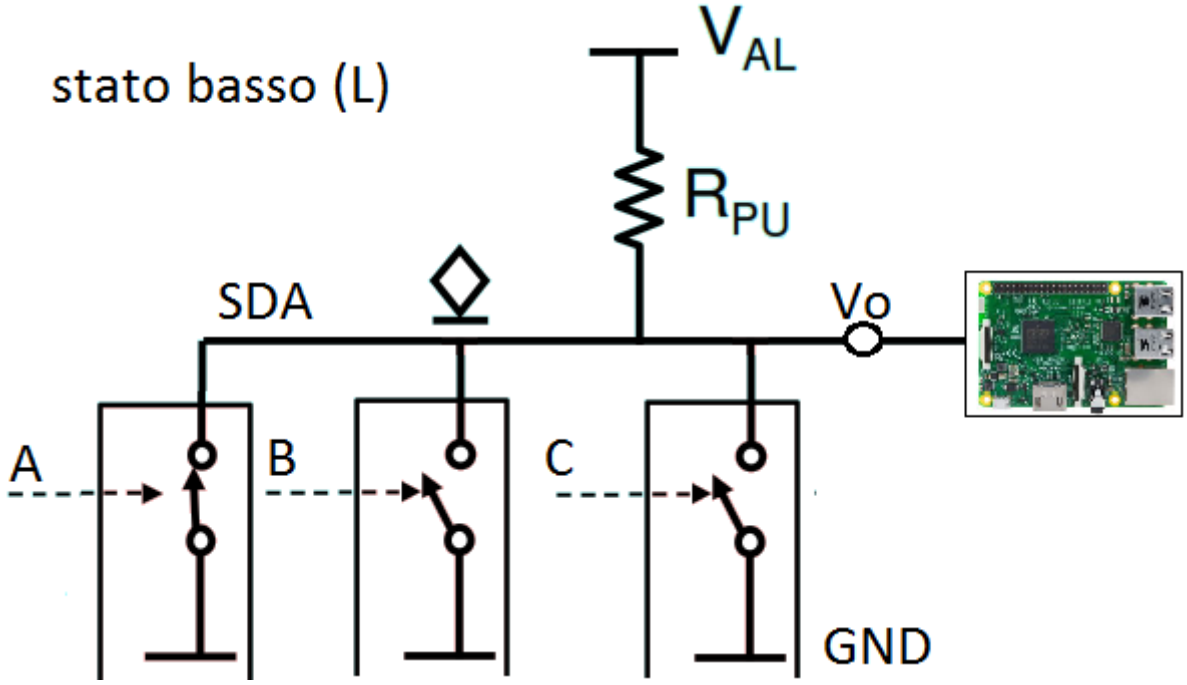


Fig.32

Il comportamento degli stati logici alto e basso è riassunto dalla funzione logica NOR di Tab.3 nella quale sono riportati i tre device slave A, B e C anche se spesso di parla di Wired OR ad indicare che il device slave è attivo quando l'interruttore logico è chiuso verso massa pilotando la linea con uno stato logico attivo basso (L), mentre con il device slave inattivo l'interruttore logico è aperto e grazie alla R_{PU} lo stato logico inattivo è alto (H).

A	B	C	NOR
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	1	0

Tab.3

Il discorso fatto fino a questo momento di compatibilità logica degli stati per il bus dati SDA vale anche per il bus del clock SCL, come riporta in sintesi la Fig.33 nella quale è possibile utilizzare due valori di R_{PU} diversi. La figura sottostante è generica, quindi volendo utilizzare come master il Raspberry conviene impostare V_{CC}=+3.3V. Nell'esempio il solo Device 2 è attivo alla ricezione, mentre il master trasmette.

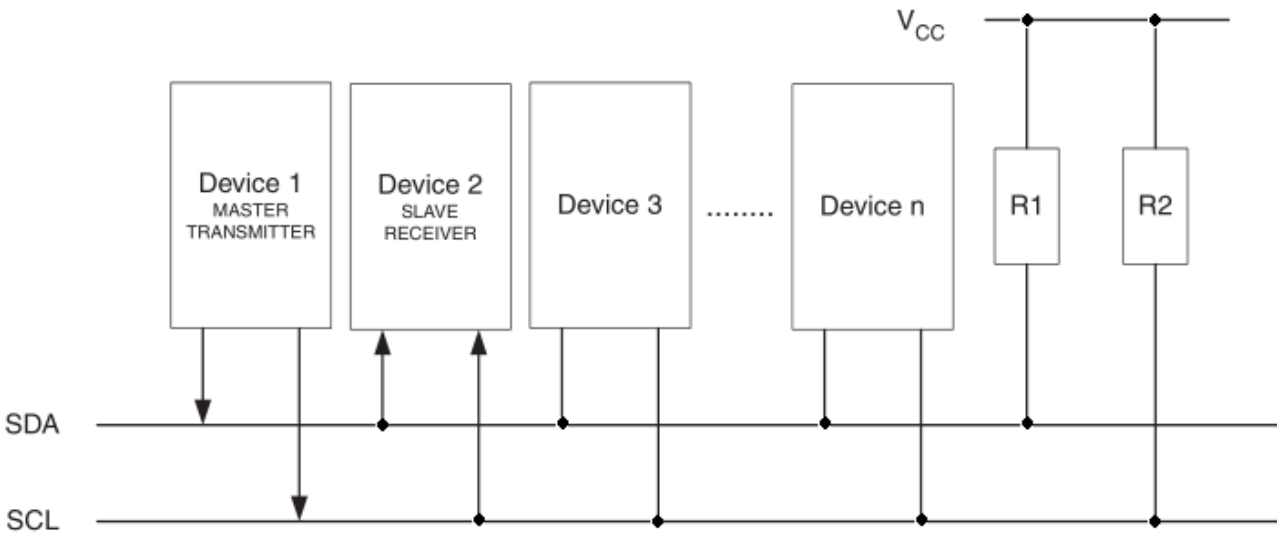


Fig.33

I^2C è il protocollo seriale sincrono complesso anche se si potrebbe pensare il contrario visto la presenza di sole due linee, una per i dati SDA ed un per il clock SCL. Ha una velocità massima di trasmissione di 400Kbps, anche se per alimentazioni a +3.3V non si va oltre i 100Kbps. Il canale di trasmissione dei dati funziona in modalità half-duplex, ossia o si legge o si trasmette tra master e slave, inoltre non è necessario nessun tipo di buffer per ospitare i dati. E' necessario ricordarsi delle problematiche di compatibilità degli stati logici tramite l'impiego di un resistore di pull-up R_{PU}. La trasmissione dei dati con I^2C possiede un protocollo di handshake con il quale è possibile correggere gli errori di trasmissione. Questo in sostanza riassume le caratteristiche base del funzionamento del protocollo I^2C.

SPI è un protocollo seriale sincrono inventato da Motorola molto semplice rispetto a I^2C, anche se in primo acchito potrebbe sembrare più complesso per via di un numero maggiore di linee che ne rende il cablaggio più articolato. Da un punto di vista elettrico necessita di quattro linee come si vede chiarimento dalla Fig.34 e offre la possibilità di trasmissioni full duplex tra master e slave attivo.

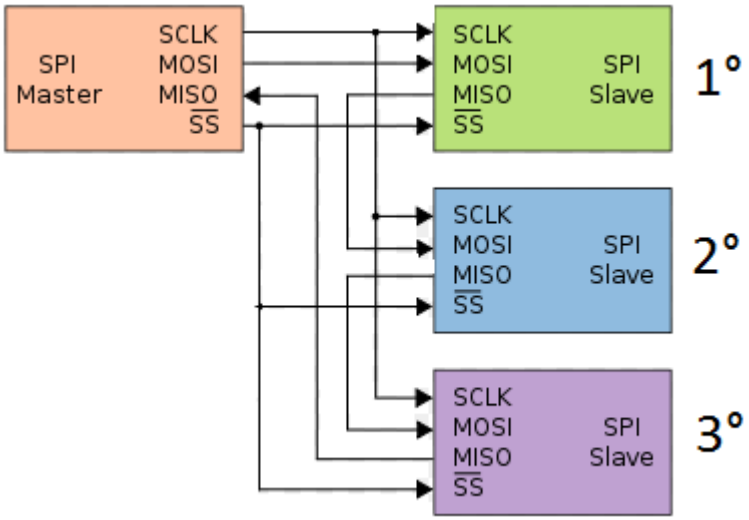


Fig.34

Il vantaggio rispetto a I^2C è la possibilità di pilotare i device slave con un clock maggiore, fino ad arrivare a 2MHz con un'alimentazione di +3.3V nel caso di Pi, ma si potrebbe arrivare anche ai 4MHz per una classica alimentazione in continua di +5V o addirittura fino a 50MHz in determinati micro controllori. SPI permette la gestione di dispositivi complessi, di conseguenza necessità obbligatoriamente sia lato master che lato slave di registri buffer

circolari a scorrimento per ospitare i dati, che possono venire realizzati in hardware come accade su molti convertitori A/D, oppure in software come avviene sul Raspberry Pi. La selezione dello slave da parte del master avviene tramite la linea SS (Slave Select), chiamata anche CS (Chip Select), sulla quale viene posto il livello logico basso (L) o direttamente 0V, visto che molti device hanno il pin SS attivo basso. Questo unico collegamento può portare ad avere tutti gli slave attivi, ma solo l'ultimo slave numero 3 della catena comunica i dati al master tramite la linea MISO (Master Input Slave Output). Il master a sua volta trasmette i dati allo slave numero 1 della catena tramite la linea MOSI (Master Output Slave Input). Il segnale di clock SCKL è comune a tutti essendo tale tecnologia di tipo sincrono. La Fig.25 indica in viola i pin 19 (MOSI), 21 (MISO), 23 (SCLK) del protocollo SPI e due canali CS0 e CS1 per il Chip Select sui pin 24 e 26 ad indicare una configurazione diversa da quella a cascata (Daisy Chain) di Fig.34. Nella Fig.35 il master possiede tra uscite indipendenti SS1, SS2 e SS3 attive basse per l'attivazione di un determinato slave. Osservando bene l'immagine i due canali MISO e MOSI sono condivisi, quindi un'attivazione errata di due o più slave in lettura può generare sul canale MISO una sovrapposizione dei segnali, con l'impossibilità di interpretare in modo corretto i dati. Il Raspberry Pi utilizza questa logica, quindi porre la massima attenzione nel momento in cui si decide di impiegare una simile configurazione.

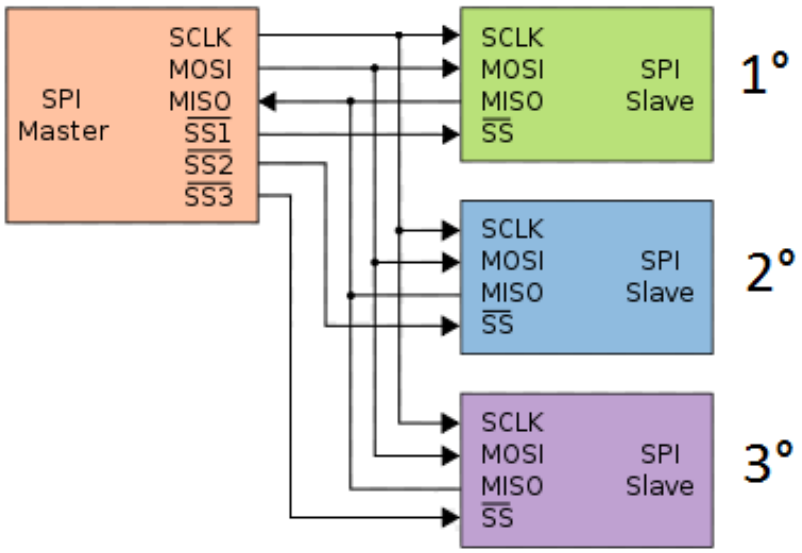


Fig.35

L'interazione tra i canali MOSI e MISO del master e slave, è soggetta all'impiego di registri a scorrimento circolari necessari alla memorizzazione dei bit. La Fig.36 indica chiaramente il funzionamento in presenza di un solo slave. I device esterni hanno spesso implementato questi registri nel hardware della scheda, mentre i sistemi embedded tendono, per questioni di costi e spazio, ad implementarli via software dal sistema operativo, ma ovviamente non è un regola.

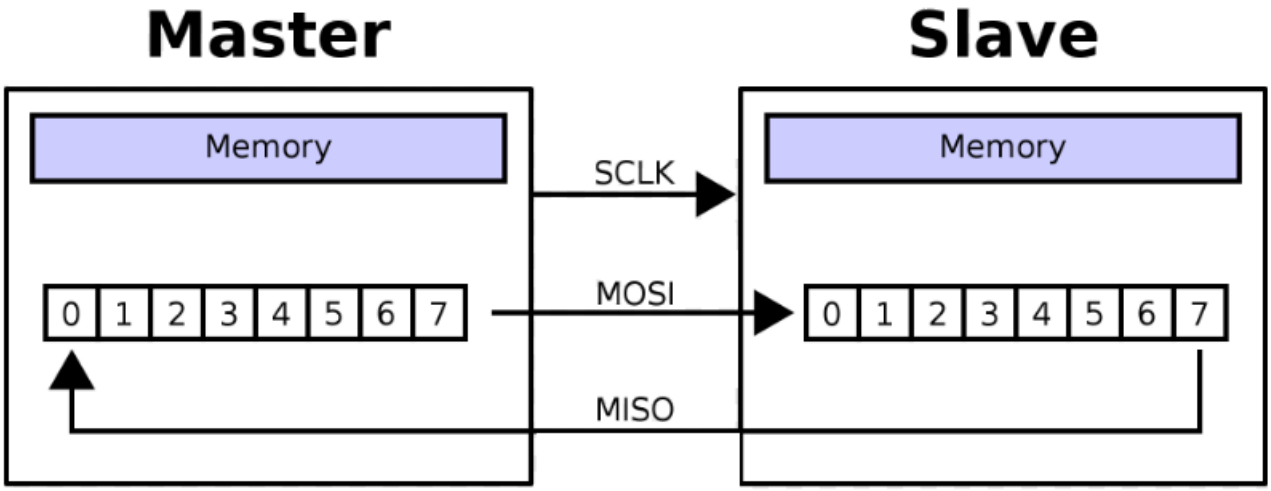


Fig.36

Il master è l'unico elemento del sistema SPI che imposta il clock, così che si possa avere la trasmissione seriale dei bit tra master e slave, ma anche in contemporanea la ricezione di bit dallo slave al master sempre che lo slave abbia necessità di inviare informazioni al master. La configurazione che si ottiene è un anello sincrono di trasmissione full duplex dove i dati sono veicolati in presenza del clock, che in molti device SPI è attivo basso, anche se conviene riferirsi alla documentazione del periferico e del micro controllore. Lo slave in SPI non gestisce assolutamente il clock, cosa che avviene in I^2C quando è lo slave che trasmette impostando il clock SCL per il master in ascolto. Per tale ragione in I^2C il clock deve essere molto preciso, mentre in SPI ogni slave si adatta anche a cambi improvvisi del segnale di clock imposti dal master, quindi un semplice oscillatore RC è più che sufficiente come generatore di clock in SPI. E' interessante tornare sul ruolo dei due registri a scorrimento circolari, che il più delle volte sono a 8bit. Il software di trasmissione presente sul master, invia i bit in parallelo nel registro a scorrimento, il quale a sua volta invia in modalità seriale i bit, partendo dal bit più significativo MSB, tramite la linea MOSI, allo slave, il quale inserisci i bit nel proprio registro a scorrimento. L'applicazione di

ricezione lato slave preleva i dati in parallelo per poi gestirli. Il comportamento dello shift register del master è chiamato PISO (Parallel In Serial Out), mentre quello dello slave è chiamato SIPO (Serial In Parallel Out). La Fig.37 riassume quanto esposto.

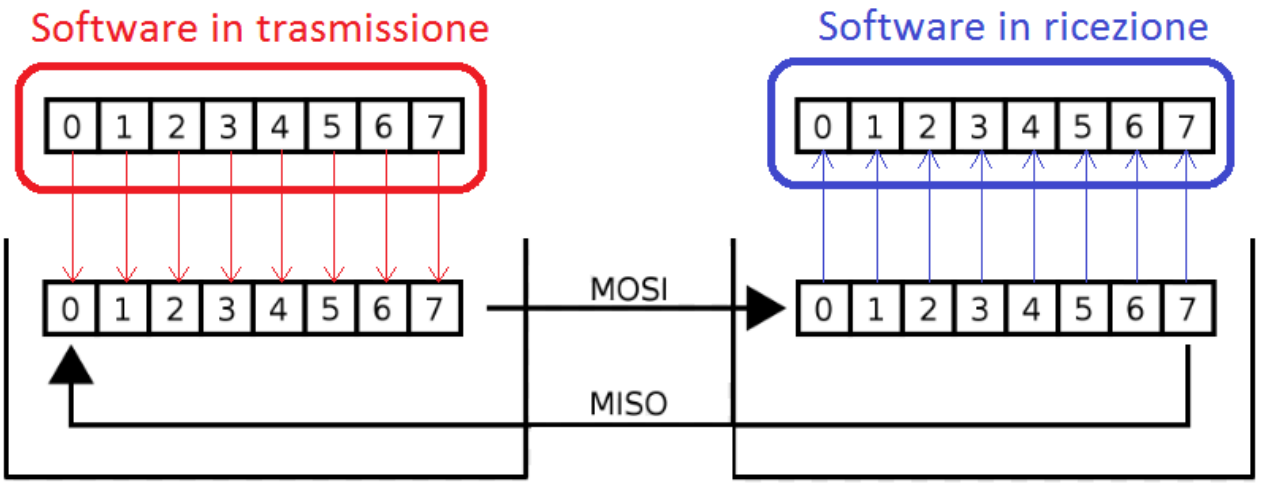


Fig.37

Prima di riassumere le differenze e gli usi di SPI e I²C, conviene vedere in che modo funziona la temporizzazione, così che sia chiara la modalità con cui i dati vengono inviati da master e slave, ed eventualmente, anche da slave a master ottenendo così un anello sincrono di trasmissione full duplex.

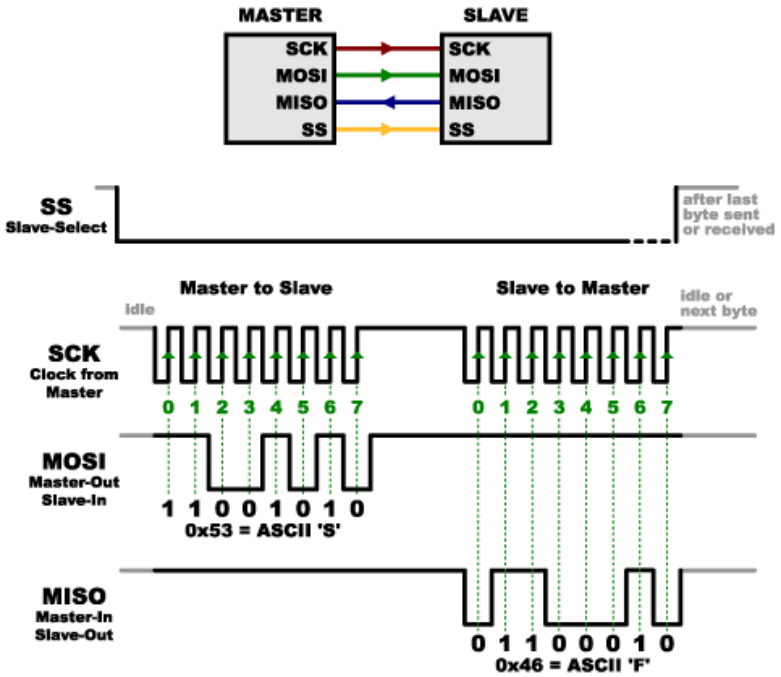


Fig.38

Dalla Fig.38 si vede una tipica configurazione con un solo slave e la sua attivazione grazie all'invio del segnale attivo basso SS (Slave Select) che però va verificato dalla documentazione del dispositivo esterno visto che potrebbe anche essere stato progettato per ricevere un segnale attivo alto. Sovente il SS (chiamato anche CS) è attivo basso. Il clock impostato dal solo master lavora sul fronte di salita, come indicano le frecce verdi. Ad ogni impulso di clock, il bit presente nel registro a scorrimento del master viene veicolato, come valore di tensione di linea del MOSI, nel registro a scorrimento dello slave. La sequenza di 8bit "11001010" viene inviata completamente dopo 8 colpi di clock sul fronte di salita. Identico discorso avviene per la linea MISO, qualora il master attenda bit dallo slave. Visto che la trasmissione del bit, sia esso da master a slave e viceversa avviene sul fronte del clock imposto dallo stesso master, non è richiesto nessun tipo di hardware che fornisca un clock di precisione, per tale ragione, se necessario, il clock può venire variato a piacimento in tempo reale.

Dopo questa panoramica del protocollo SPI, l'utente può chiedersi le differenze con I^2C, anche se qualche indicazione è già stata data nella trattazione, ma sopratutto quando utilizzare una tecnologia piuttosto che l'altra.

SPI e I^2C sono due protocolli seriali sincroni, anche se volendo essere rigorosi SPI non meriterebbe il titolo di protocollo perché è privo di meccanismi di handshake tramite i quali è possibile risolvere gli errori di trasmissione, caratteristica invece posseduta da I^2C che lo rende, di conseguenza, una tecnologia più complessa. SPI utilizza 4 linee per una configurazione daisy-chain, mentre I^2C solamente 2, ma questo fatto è fuorviante perché le linee SS, MISO, MOSI e SCK del SPI sono uni-direzionali, mentre la SDA e SCL del I^2C sono bidirezionali, il che comporta un grado di complessità maggiore a sfavore di I^2C. Proprio per questo fatto la velocità di clock è nettamente più bassa nell' I^2C, aggirandosi indicativamente sui 4MHz per valori d'alimentazione di almeno +5V, inoltre il clock, a causa della bi-direzionalità del SCL, deve essere molto preciso. Nel SPI il clock può raggiungere i 50MHz ed è uni-direzionale, ossia è il solo master che lo imposta a favore dello slave, senza la necessità di generatori di clock precisi, quindi un semplice oscillatore RC è più che sufficiente. In SPI il clock può anche essere variato da slave a slave così da adattarsi a diverse periferiche. SPI permette la trasmissione full-duplex, mentre I^2C solo quella half-duplex, quindi SPI è più indicato per sistemi di streaming di byte, dove una continua interazione di dati tra master e slave e viceversa è l'obiettivo desiderato. In presenza di molti slave l'utilizzo di SPI è problematico, visto che o si utilizza un master che

ha più pin CS (Chip Select) come in Fig.35, e quindi si avrà obbligatoriamente un numero massimo di slave, o si utilizza la configurazione daisy-chain di Fig.34 che però deve essere realizzata in modo rigoroso così da chiudere l'anello sincrono di trasmissione. Con il termine rigoroso si intende che per avere un anello sincrono di trasmissione tutti i device slave devo obbligatoriamente essere attivi, quindi perché ciò accada tutti gli slave devono attivarsi sul medesimo stato basso (L) del CS. Il protocollo I^2C è più adatto in presenza di configurazioni con molti slave, visto che la selezione di un dato slave avviene semplicemente inviando un indirizzo sul bus dati SDA, a discapito della velocità di trasmissione. Formalmente l'invio di tale indirizzo è visto come overhead. I^2C riesce a capire se lo slave selezionato è attivo oppure no grazie alla presenza del handshake, SPI invece non è in grado di rilevarlo. I^2C permette l'invio di informazioni più grandi del byte, anche flussi molto lunghi di informazioni, ma in modalità half-duplex. La tecnologia SPI non necessità di nessun resistore di pull-up ai fini dell'adattamento logico derivato dall'alta impedenza, mentre I^2C ne ha bisogno col fine di avere due stati logici ben definiti e quindi correttamente interpretati dal master. La presenza del resistore di pull-up R_{PU} comporta un ovvio consumo di energia. La scrittura di un software in I^2C è più elaborata in quanto si deve implementare il meccanismo di handshake che in SPI è assente.

Rimanendo sul generico sistema SPI e I^2C, qualora sia master che slave non possiedano al proprio interno il modulo di gestione per la trasmissione sincrona, è necessario intervenire via software per la realizzazione di un simulatore, quindi conviene utilizzare un sistema SPI dato che il software che dovrà venire scritto sarà inevitabilmente più semplice, visto che I^2C dovrà gestire la selezione dello slave tramite indirizzo, ma anche il meccanismo di handshake.

Per quanto detto fino ad ora le situazioni più indicate per l'impiego di I^2C è in presenza di sistemi multi-slave, dove ogni dispositivo disponga di un modulo di trasmissione sincrono senza che vi sia la necessità di elevate velocità. Le situazioni più indicate per SPI è in presenza di pochissimi slave, hardware privo di moduli di trasmissione o necessità di alte velocità di trasmissione. La Tab.4 riassume le principali differenze tra SPI e I^2C.

	I²C	SPI
Numero delle linee	2 bi-direzionali	almeno 4 uni-direzionali
Clock	preciso e bi-direzionale	adattativo allo slave e non preciso
Tipo di trasmissione	half duplex	full duplex
Rilevamento slave	si tramite handshake	no
Numero slave	molti	pochi
Tipo di attivazione slave	tramite indirizzo	tramite SS (CS) attivo basso
Velocità di trasmissione	bassa	elevata
Emulazione trasmissione	complessa	semplice
Protocollo semplice	no	si

Tab.4

L'ultima tecnologia presente nel GPIO di Fig.25 è l'UART (Universal Asynchronous Receiver Transmitter), indicata in verde dai pin numero 8 UART0 TX e pin numero 10 UART0 RX che serve per trasformare i bit inviati in modalità parallela dal processore in una sequenza di bit seriali che poi vengono posti sulla linea di trasmissione TX. L'invio delle informazioni avviene senza l'ausilio del clock e il ricevente provvederà a leggere i bit prelevandoli dalla linea TX. Il ruolo di trasmettitore e ricevitore viene anche indicato spesso con i termini DTE (Data Terminal Equipment) e DCE (Data Communication Equipment). La Fig.39 mostra un collegamento seriale asincrono tra due Raspberry, chiamato anche null-modem, il quale prevede che il canale di trasmissione del primo Pi sia collegato sui pin di ricezione del secondo Pi e viceversa. Entrambi i device devono avere una massa comune per garantire equi potenzialità.

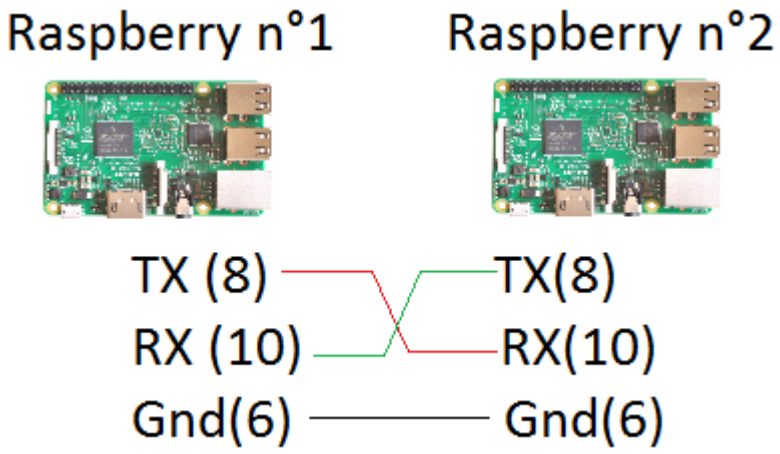
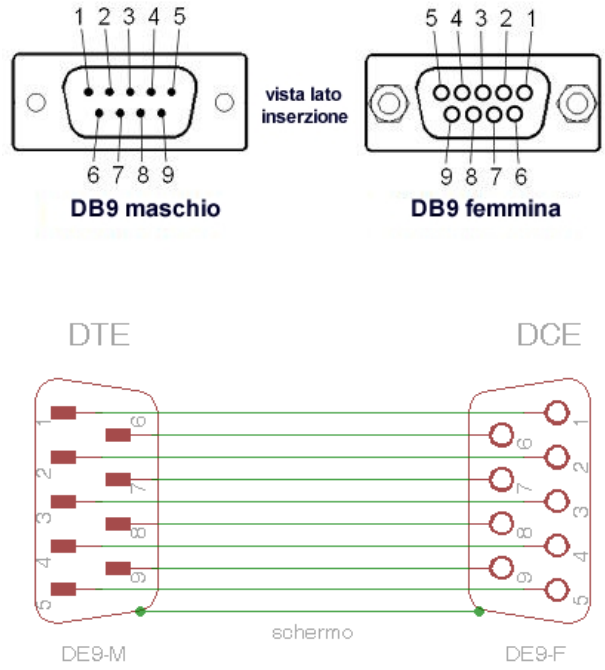


Fig.39

Questi tipo di tecnologia basata sui soli pin TX e RX non ha nessun tipo di gestione hardware del handshake, la quale, se necessaria deve venire implementata lato software. Molte tecnologie UART compatibili con lo standard EIA RS-232 permettono la gestione di handshake hardware, con un conseguente aumento dei pin rispetto a quelli presenti nel Pi e non esiste un uso standard nell'impiego pratico di tutti questi segnali visto che ogni produttore di dispositivi seriali può realizzare la propria implementazione che comporta inevitabilmente lo studio del manuale, nel caso di interazione tra dispositivi, o peggio ancora il reverse engineering. Un esempio di interazione fisica tra due dispositivi RS-232 è quella di Fig.40, che però non garantisce necessariamente che i due device possano dialogare correttamente, serve che vengano usati i medesimi bit di controllo.



lato DTE		Direzione del segnale	lato DCE	
Pin	Funzione		Pin	Funzione
1	DCD	←	1	SH
2	RXD	←	2	RXD
3	TXD	→	3	TXD
4	DTR	→	4	DTR
5	GND	—	5	GND
6	DSR	←	6	DSR
7	RTS	→	7	RTS
8	CTS	←	8	CTS
9	RI	←	9	RI

Fig.40

Il connettore di Fig.40 è un classico DE-9, che negli anni passati era presente sul mouse e sulla tastiera, mentre oggi giorno lo si usa ancora per programmare via console device che utilizzano una RS-232 come gli switch e/o i router. Dalla tabella inserita in Fig.40 si vede che i segnali hardware di handshake sono il DSR, il DTR, il DCD, il RTS, il CTS ed il RI. L'obiettivo di questo fine paragrafo non è una trattazione completa del RS-232, ma mettere in risalto l'impiego classico di connessioni DTE-DCE, ossia la comunicazione seriale a lunga distanza. La velocità di trasmissione nell'UART è inversamente

proporzionale alla distanza tra le due stazioni, quindi velocità elevate possono provocare deformazioni nei fronti di salita e discesa a causa di capacità e induttanze parassite introdotte dai cavi stessi. Per quanto possa sembrare strano lo standard EIA-RS-232 non definisce una determinata tipologia di cavo, ma solo la capacità massima del cavo e del ricevitore. Lo standard riporta che la lunghezza massima del cavo è pari a una capacità di 2500pF, che in altri termini vuol dire una distanza di 25 metri per cavo con 100pF/metro. Lo standard dei cavi di reti EIA-TIA-568B, prevede che i cavi UTP Cat.5 abbiamo 17pF/metro, il che comporta che l'impiego di una tipologia di cavo di rete per trasmissioni seriali può arrivare indicativamente ai 147 metri (2500pF/17pF), considerando nulla la capacità dei connettori e dei ricevitori. In ogni caso riducendo la velocità di ottengono distanze maggiori e diviene interessante osservare i dati della Tab.5 prodotti da uno studio della Texas Instrument sul rapporto velocità-distanza, ossia una riduzione nella frequenza di trasmissione provoca un aumento della distanza tra le stazioni.

Baud Rate [bps]	Lunghezza del cavo [m]
19200	15
9600	150
4800	300
2400	900

Tab.5

L'ultima riga è alquanto interessante, infatti 2400bps significa 300byte/s, e qualora il trasmettitore debba inviare solo poche manciate di informazioni ad intervalli di tempo laschi, il potere avere una distanza prossima ai 900 metri tra le stazioni può essere una soluzione che con SPI e I^2C è impensabile. Qualora una delle due stazioni venga impostata con una velocità diversa, la trasmissione non potrà avere avvenire, così come una diversa configurazione dei bit di controllo o dei messaggi di handshake che ogni costruttore può personalizzare a proprio vantaggio.

Capitolo 2

Microsoft Windows 10 IoT

Questo capitolo descrive le procedure per scaricare Windows 10 IoT per Raspberry, come effettuare l'installazione e i passi per una corretta configurazione.

2.1 *Introduzione a Windows 10 IoT*

La diffusione dei sistemi embedded ha spinto la casa di Redmond alla realizzazione di una versione dedicata per tale tecnologia chiamata Windows 10 IoT (Internet of Things) che possa permettere ai programmatori di alto livello di scrivere codice per applicazioni classiche, ma anche creare delle UWP (Universal Windows Platform) tramite il linguaggio C# per vari dispositivi.

2.1.1 Caratteristiche di Windows 10

Il sistema operativo Windows 10 è stato sviluppato per offrire una piattaforma universale che possa coprire una varietà di famiglia di prodotti presenti sul mercato, dal semplice smartphone, fino al pc desktop, dal server aziendale fino ai dispositivi che fanno parte del mondo IoT. Le versioni di Windows 10 sono in totale sette con costi di licenza differenti, ma solo la versione IoT è offerta gratuitamente per invogliare il mondo degli sviluppatori a scrivere le proprie applicazioni per tale sistema. La Tab.6 riassume le differenza sostanziali tra i vari sistemi.

Sistema operativo Windows 10	Caratteristiche
Home	edizione desktop per PC, tablet 2-in-1 con Cortana, Microsoft Edge, Continium e UWP
Mobile	edizione per smartphone e piccoli tablet con versione touch per Office
Pro	tutte le funzionalità della versione Home con la possibilità di inserimento in un Active Directory Domain e sfruttare il Windows Update for Business
Enterprise	tutte le funzionalità della versione Pro con l'aggiunta di specifiche per le medie e grandi imprese
Education	tutte le funzionalità della versione Enterprise
Mobile Enterprise	tutte le funzionalità della versione Mobile dedicate all'azienda
IoT Core	versione gratuita per i device IoT

Tab.6

Dalla tabella si capisce che l'obiettivo principale in casa Redmond è avere un sistema operativo multi piattaforma che permetta alle persone di non essere più vincolate a determinati device, ma di poter permettere l'interazione tra mobile phone e pc, tra tablet e notebook fino ai micro controllori per IoT. La logica di progettazione di Windows 10 si basa quindi sul concetto di "mobile first - cloud first".

2.1.2 Download Windows 10 IoT

Windows 10 IoT Core è scaricabile dal sito ufficiale https://developer.microsoft.com/it-it/windows/iot come si veda dalla Fig.41.

Fig.41

Lo step successivo è l'accesso al portale tramite il pulsante "Accedi" come da Fig.42, che comporta la creazione di un account qualora l'utente non abbia mai proceduto ad una prima registrazione o l'inserimento delle proprie credenziali. Si veda la Fig.43.

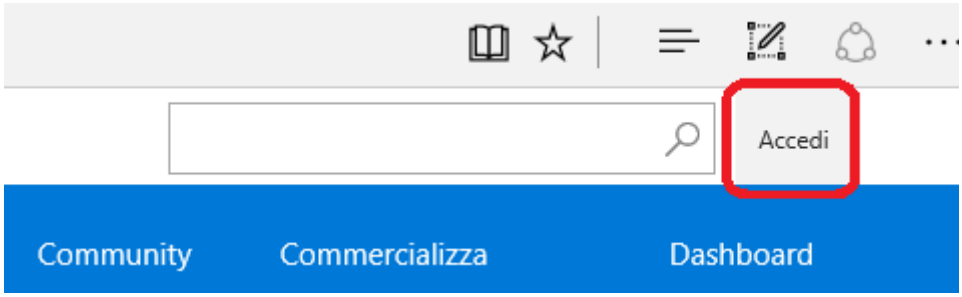


Fig.42

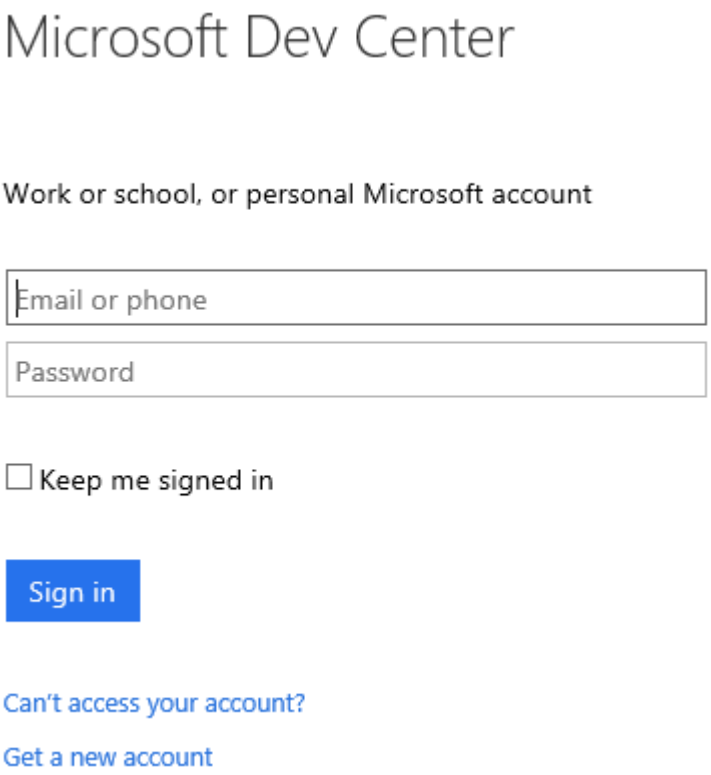


Fig.43

L'accesso al sistema è immediato con il re-indirizzamento alla pagina iniziale dove appare chiaramente nell'angolo in alto a destra il proprio account, come si vede chiaramente nella Fig.44.

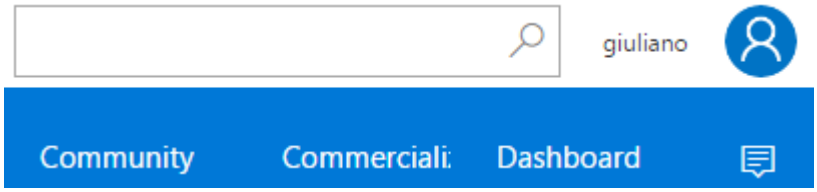


Fig.44

Con un semplice click sulla voce "Downloads" si entra nella sezione dove è possibile scaricare la dashboard di Windows 10 IoT che è un software da installare su un sistema Windows 10 Home o superiore con il quale si possono amministrare più schede embedded IoT col fine di configurarle ai propri scopi. Si veda la Fig.45.

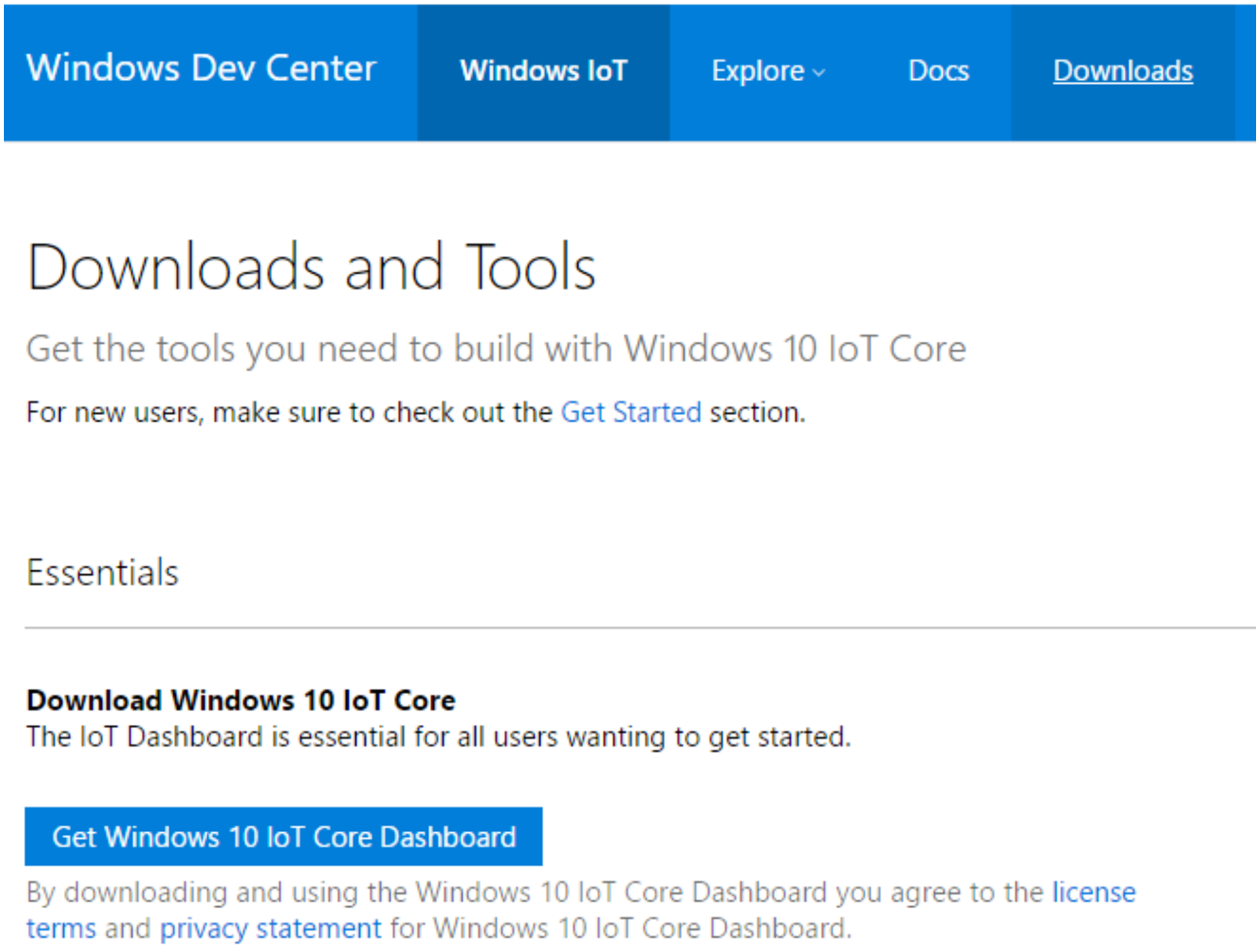


Fig.45

Procedere allo scaricamento della dashboard premendo il pulsante "Get Windows 10 IoT Core Dashboard" della figura precedente. Il file che viene scaricato è un "setup.exe" di piccole dimensioni il quale è immediatamente installato sul sistema Windows 10 come da Fig.46.

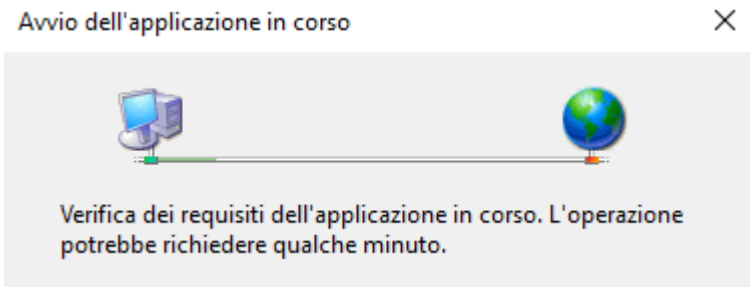


Fig.46

La procedura di installazione richiede un collegamento alla rete col fine di scaricare dei file necessari all'esecuzione della stessa dashboard, come si evince da Fig.47.

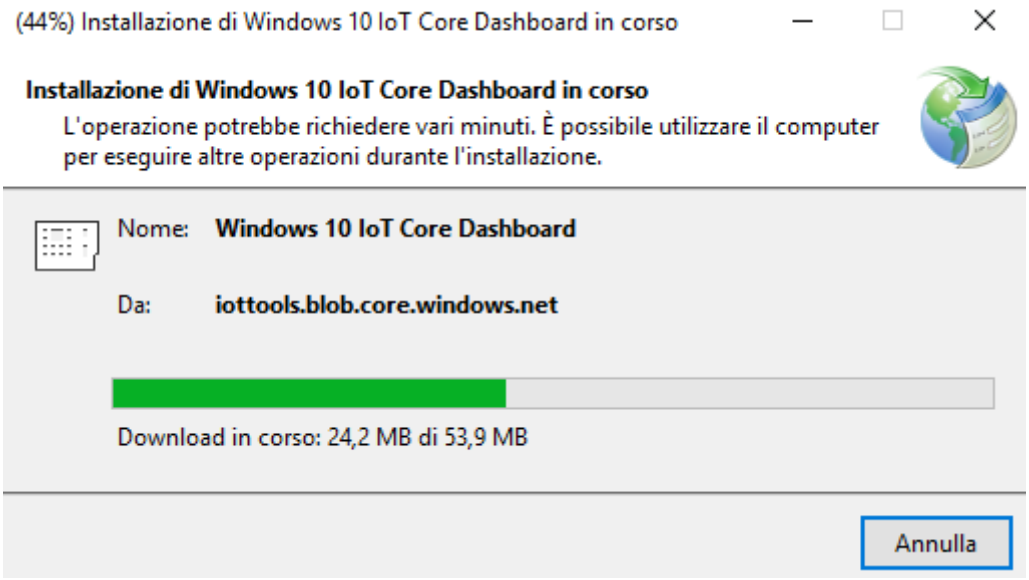


Fig.47

Per avviare successivamente la dashboard è necessario riavviare il file "setup.exe", il quale avendo già scaricato in precedenza i files, avvierà immediatamente l'applicazione. La presentazione della dashboard con la quale interagire con un pool di device IoT è ben visibile in Fig.48.

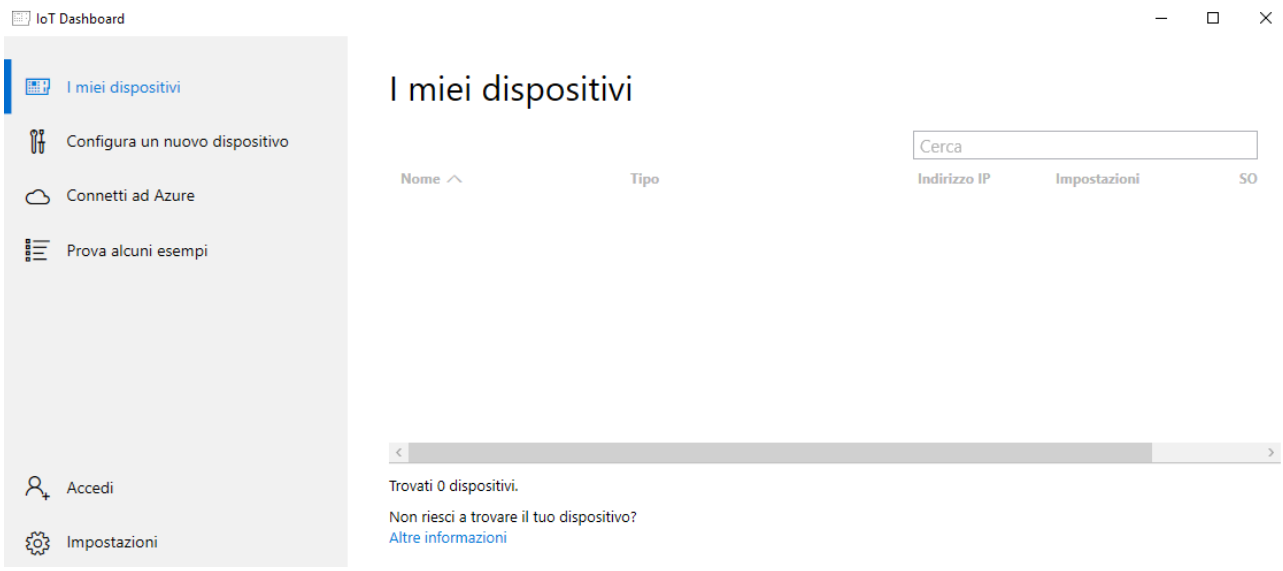


Fig.48

2.2 *Installazione*

La procedura di installazione è banale grazie all'ultima release della dashboard, anche se vi potrebbero essere del problemi legati al tipo di scheda micro SD utilizzata.

2.2.1 Micro SD

Il primo passo è procurarsi una scheda micro SD (eventualmente con supporto di espansione SD) da almeno 8GB di classe 10, così che possa essere assicurata una velocità di scrittura minima di 10MB/s, visto che la micro SD fungerà da memoria secondaria per archiviare in modo permanente il sistema operativo e i vari tipi di applicazioni. La Fig.49 mostra un esempio di micro SD da 32GB di classe 10 e l'adattatore presente nella medesima confezione. Procedere all'inserimento della scheda nel lettore del pc o notebook e verificare che il sistema operativo Windows 10 riconosca correttamente la scheda. Nel caso della Fig.50 la micro SD da 32GB è stata correttamente riconosciuta come device di periferica "E:\". In caso contrario potrebbe esserci un problema sul filesystem della SD, in tale circostanza conviene procedere alla rimozione delle unità logiche presenti sulla SD ed effettuare la formattazione con FAT32, tramite "Gestione disco" o utility ad hoc.

Fig.49

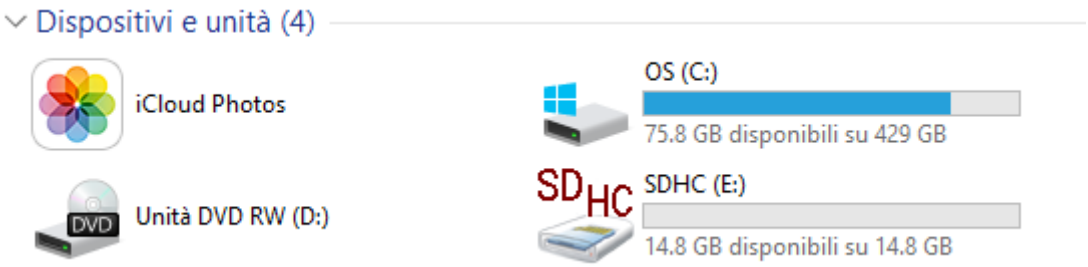


Fig.50

2.2.2 Installazione online dalla dashboard

Tramite la dashboard di Fig.51, selezionare nella voce "Tipo di dispositivo" l'opzione "Raspberry Pi 2 & 3" e come "Build sistema operativo" la voce "Windows 10 IoT Core". La micro SD dovrebbe venire riconosciuta automaticamente nella sezione "Unità", in caso contrario verificare il corretto riconoscimento da parte del sistema operativo come da Fig.50. Infine non resta che assegnare un nome al dispositivo ed impostare una password significativa da utilizzare per la gestione del dispositivo via web. Il nome host scelto è "StazioneMeteo". Il passo finale è accettare le condizioni di licenza del software e avviare la creazione dell'immagine sulla micro SD premendo il pulsante "Scarica e installa" che si trova in fondo alla Fig.51 e che per chiarezza viene riportato nella Fig.52.

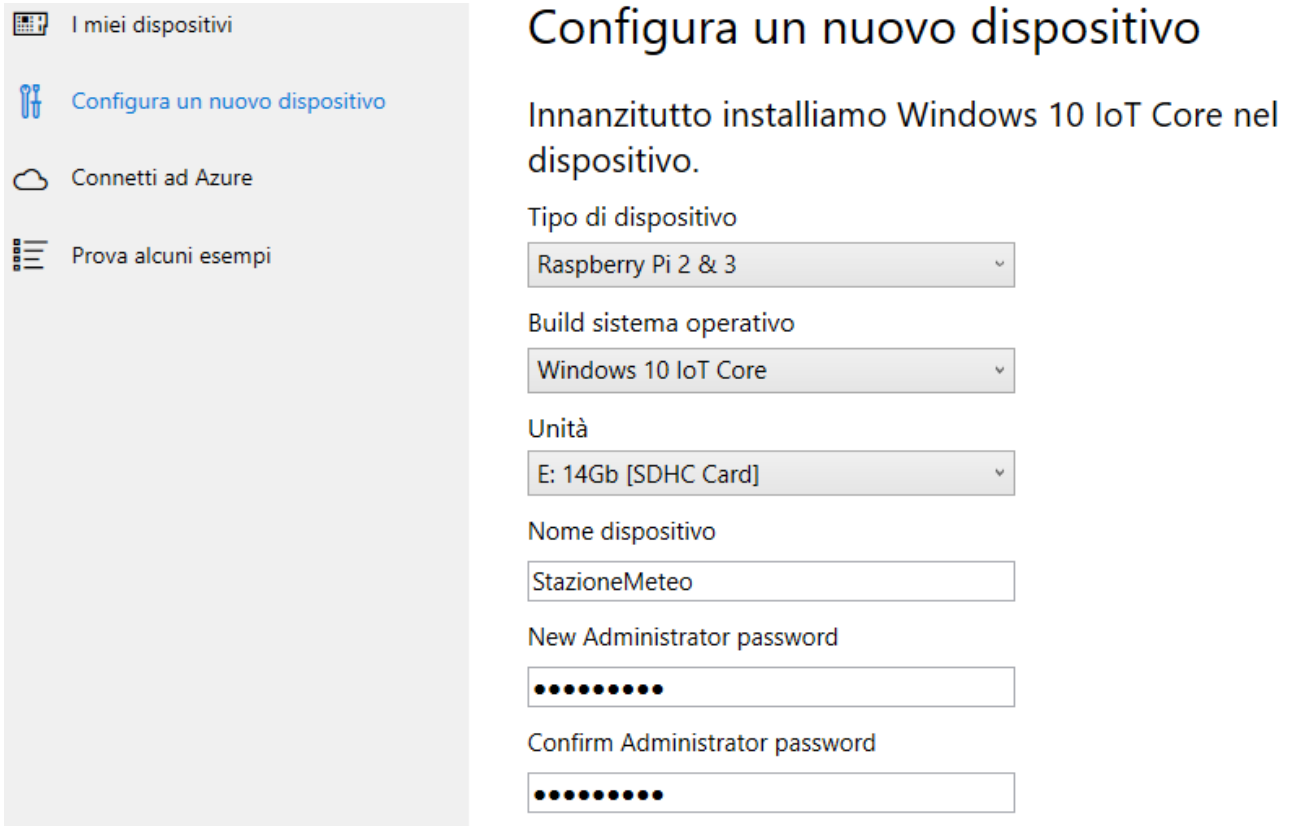


Fig.51

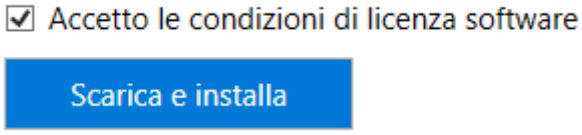


Fig.52

Il procedimento di creazione chiede conferma prima di continuare come si evince dalla Fig.53, dopo il quale parte il download del core direttamente in dashboard, Fig.54

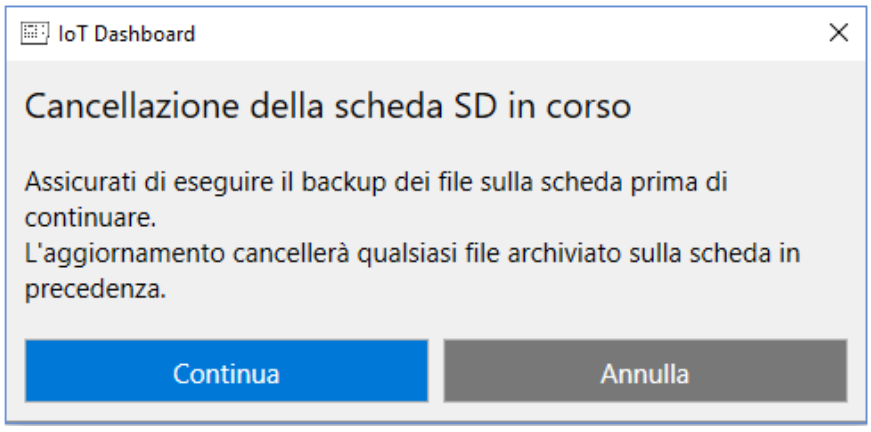


Fig.53

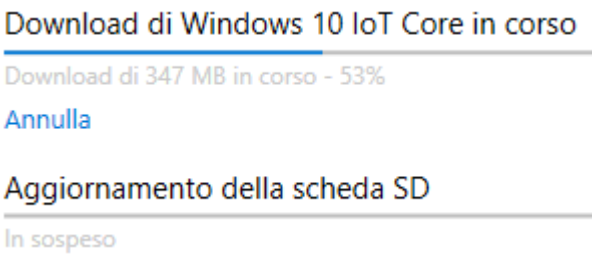


Fig.54

Terminato il download parte la decompressione del file, come da Fig.55, Windows 10 chiede l'autorizzazione per la copia dell'immagine sulla micro SD tramite l'utility di sistema "dism.exe" avviata automaticamente con i privilegi di amministratore. L'operazione di creazione è visibile in Fig.56.

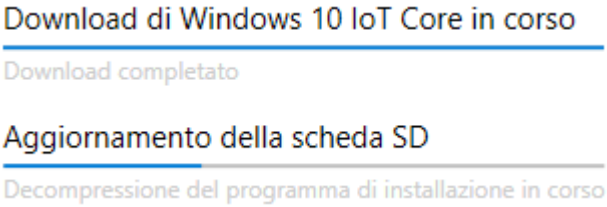


Fig.55

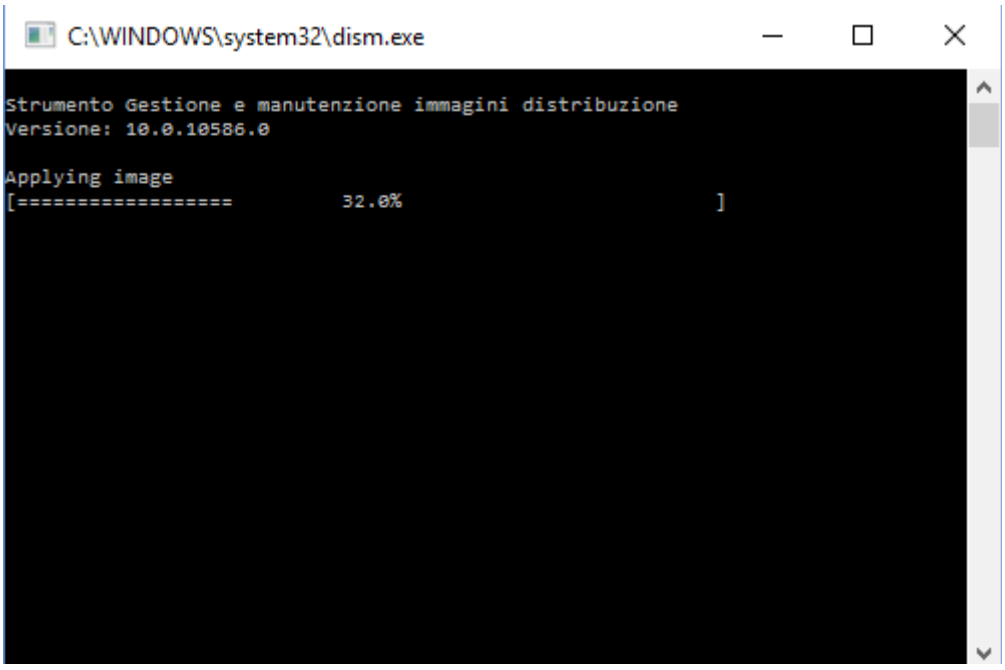


Fig.56

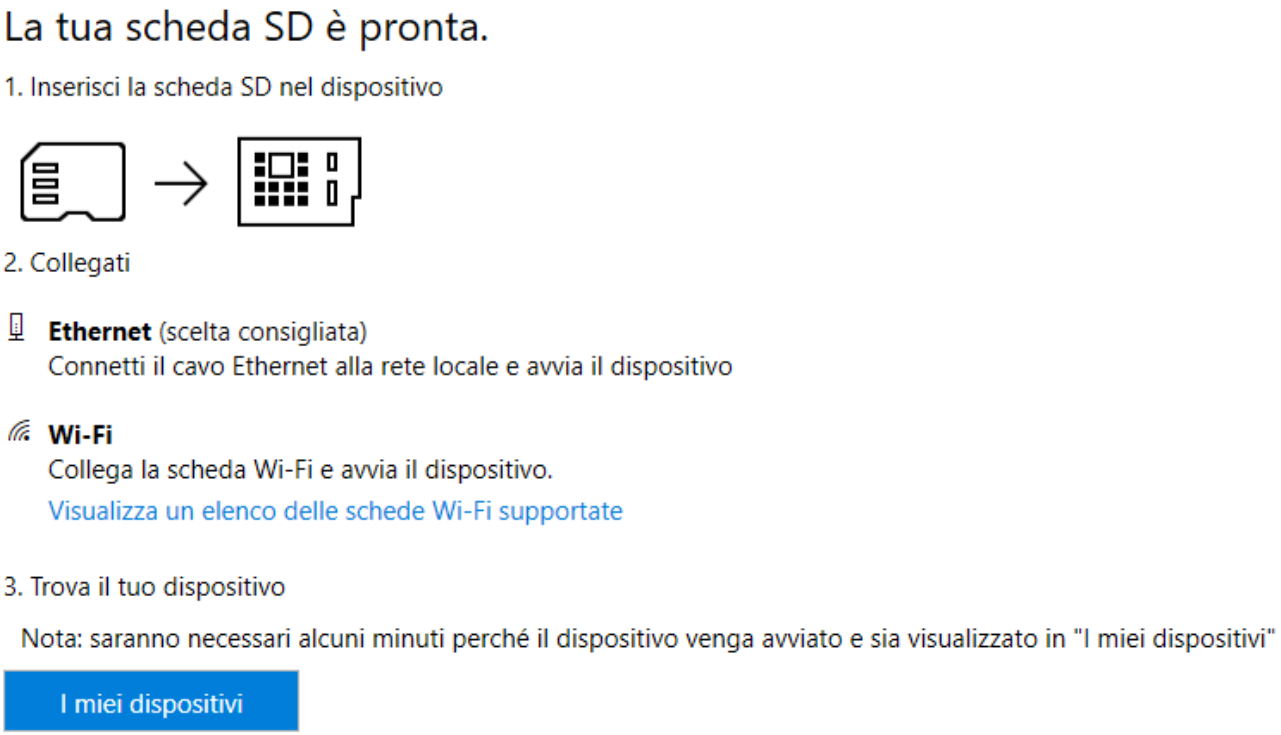


Fig.57

Terminato l'operazione di copia dell'immagine la Fig.57 avverte di inserire la micro SD nel dispositivo Raspberry collegandolo alla rete ethernet per la configurazione via web. Inserendo nuove micro SD e ripetendo la procedura con la dashboard, la fase di download verrà saltata e si procederà direttamente alla copia.

2.3 *Configurazione*

La configurazione può essere fatta in modo sommario collegando direttamente il device Pi3 ad un display o gestirlo in modo più preciso e semplice in modo remoto via web.

2.3.1 Connessione diretta

La connessione diretta può essere fatta collegando direttamente un monitor PC o una TV tramite la porta HDMI oppure tramite un display 7" da collegare alla porta DSI di Fig.24. Ovviamente è necessario l'impiego di mouse e tastiera USB, per una gestione in stile personal computer. La connettività di rete con Pi 3 può sfruttare la rete WiFi 802.11n oppure le rete Fast Ethernet 802.3u, in tale caso è necessario collegare il cavo al connettore RJ45 del Raspberry e assicurarsi, come indicato in Fig.57, che la micro SD sia

correttamente inserita. Nella trattazione viene utilizzata una connessione HDMI ad una TV 48" e una connessione di rete 802.3. Si veda la Fig.58.

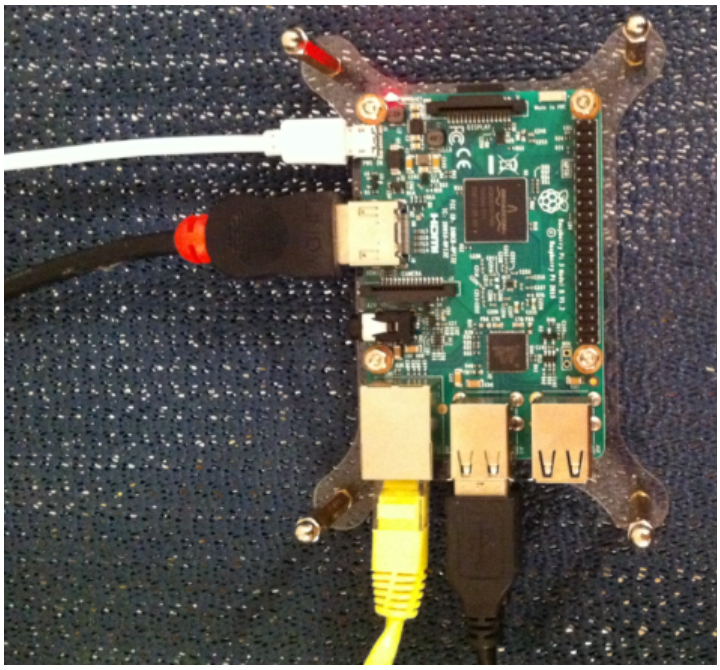


Fig.58

Una volta connessa l'alimentazione da 2.5A, parte il boot di Windows 10 IoT con il classico logo, come si vede da Fig.59.

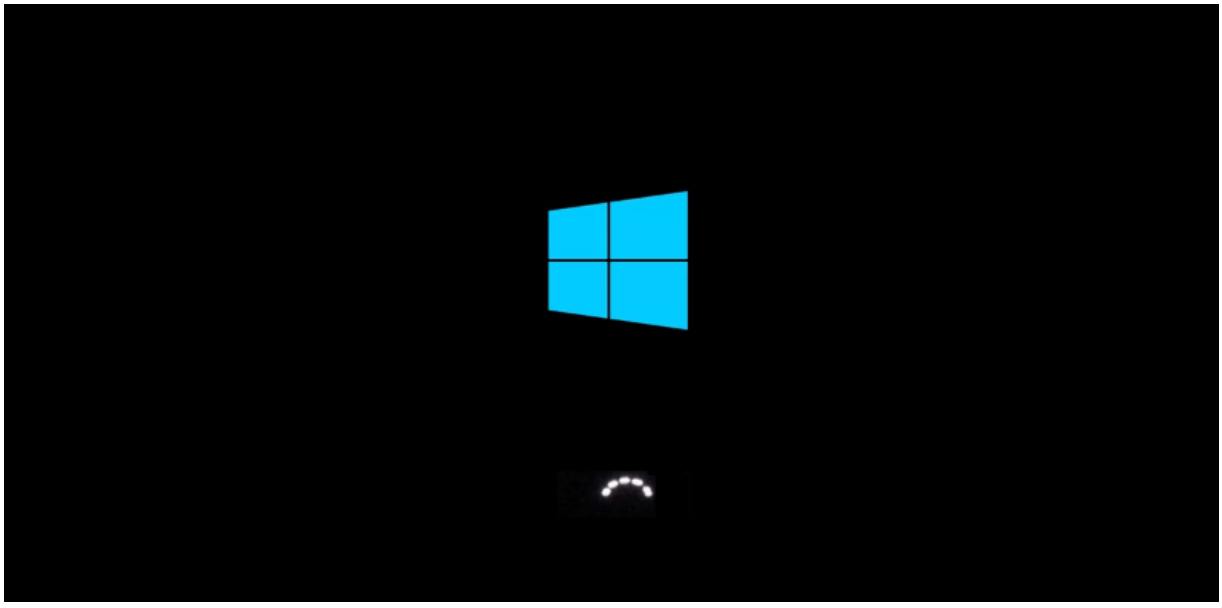


Fig.59

Il logo Windows si modifica ad indicare la versione IoT come da Fig.60.

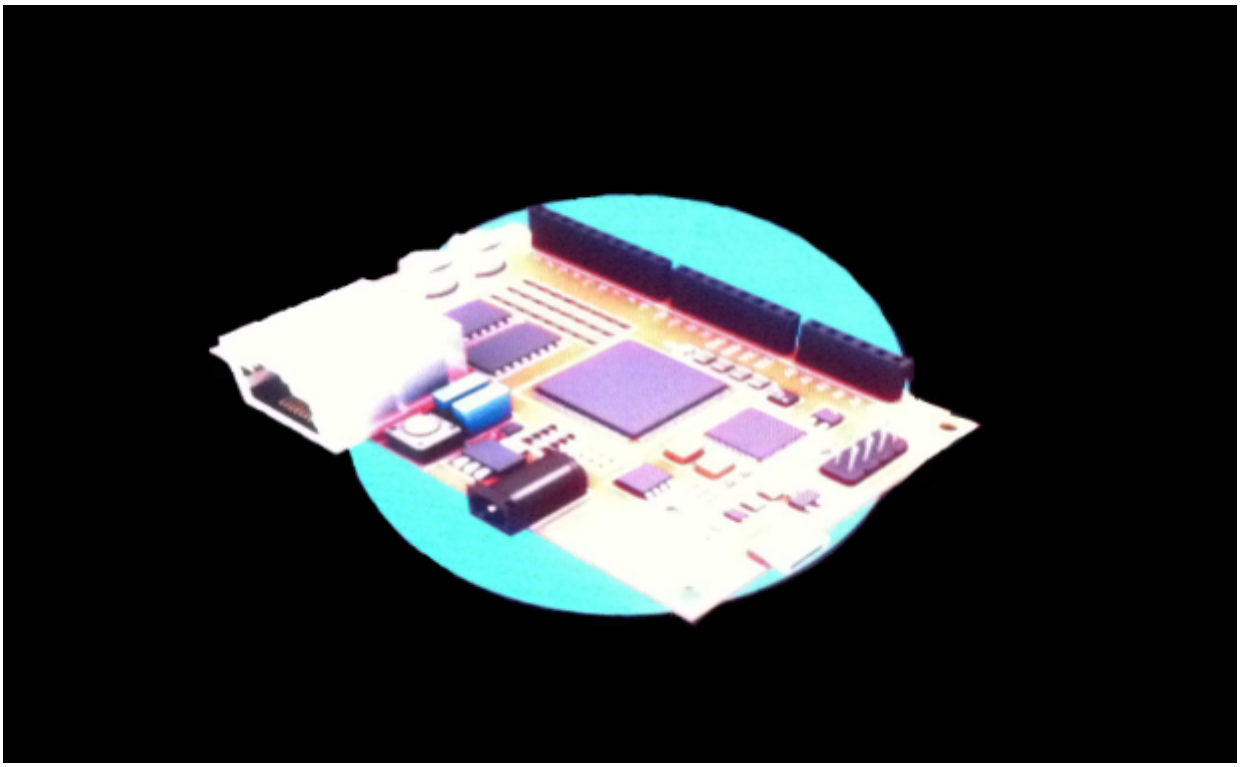


Fig.60

Terminato il boot vengono riportate le informazioni del dispositivo come si evince dalla Fig.61.

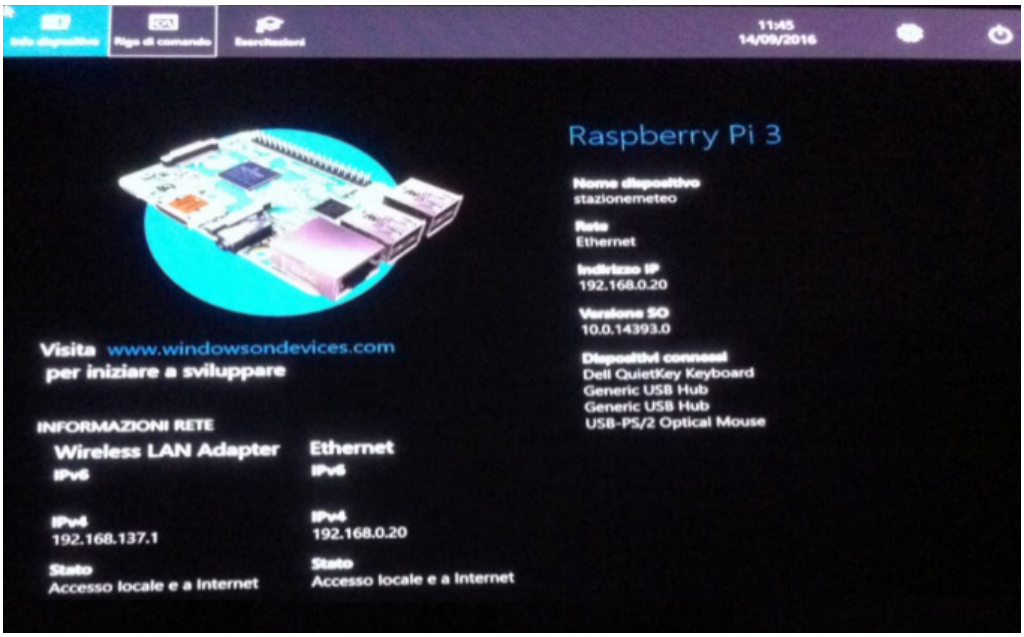


Fig.61

Il menù a video non offre molte opzioni di configurazione, ma sicuramente l'impostazione della lingua e del layout è uno step che si trova immediatamente, come nella Fig.62.

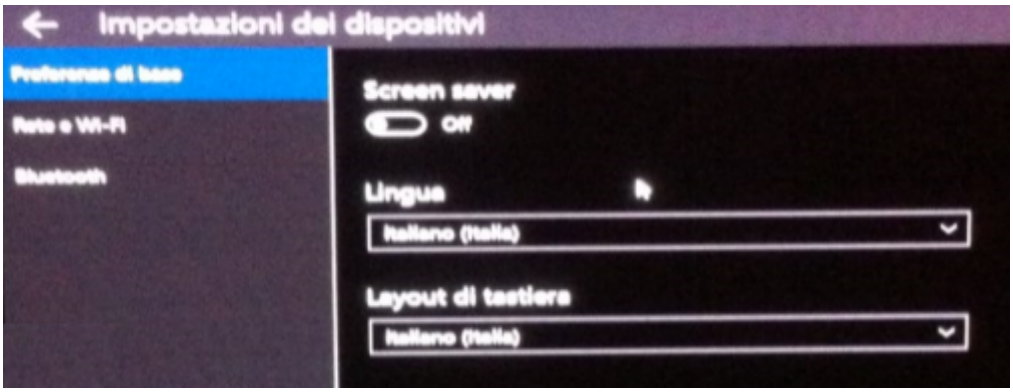


Fig.62

Dopo queste semplici impostazione è possibile effettuare lo shutdown della piattaforma, come da Fig.63, oppure lasciare connesso il device ed accedere remotamente all'indirizzo IP configurato tramite DHCP.

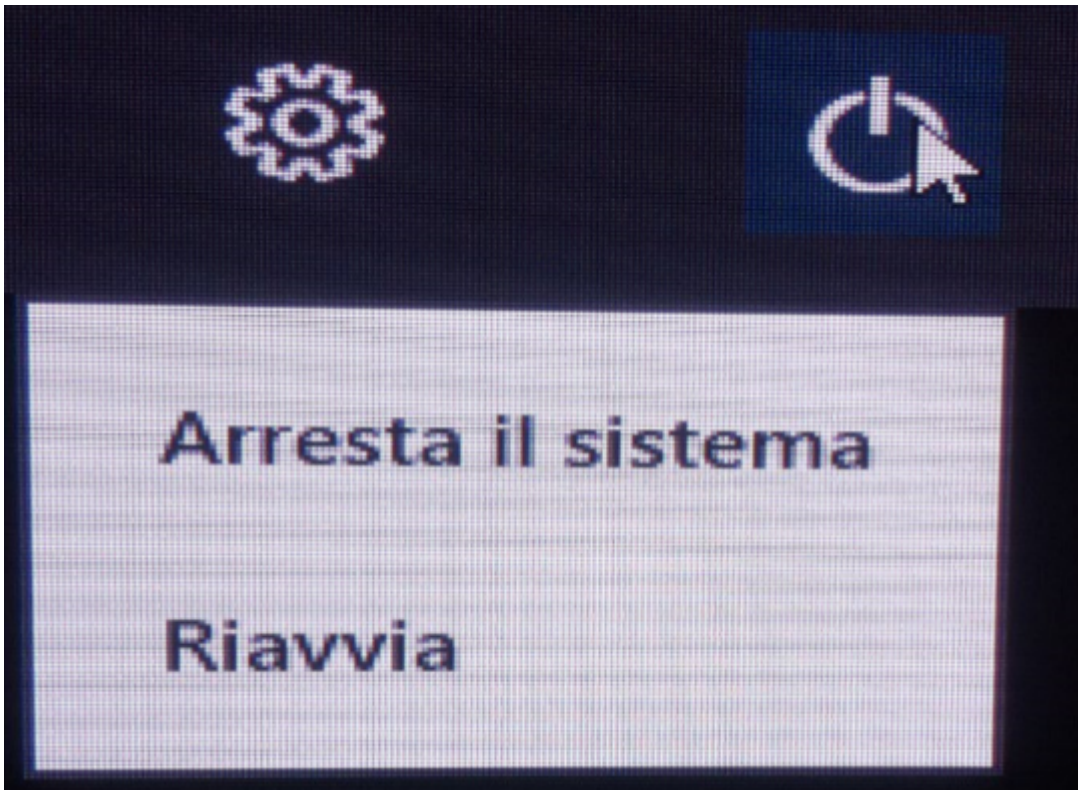


Fig.63

2.3.2 Connessione remota

La connessione diretta permette solo delle semplici impostazioni di base, quindi la scelta più logica e migliore è accedere remotamente al Raspberry tramite browser, procedura che in qualsiasi momento permette anche di aggiungere e togliere nuove UWP. Per il controllo remoto ad uno o più device è saggio utilizzare la dashboard di Fig.48, la quale può impiegare qualche istante di tempo per la rilevazione del Pi se questo è stato appena acceso. In Fig.64 è visibile il rilevamento del Raspberry Pi "StazioneMeteo" con il relativo indirizzo IP acquisito via DHCP, che nel caso è 192.168.0.20. La release del core di Windows 10 IoT è la 14393, ultima Build disponibile scaricata in automatico dalla dashboard.

Fig.64

Selezionare il device da configurare e col tasto destro del mouse selezionare la voce di menù "Apri in Device Portal" come da Fig.65.

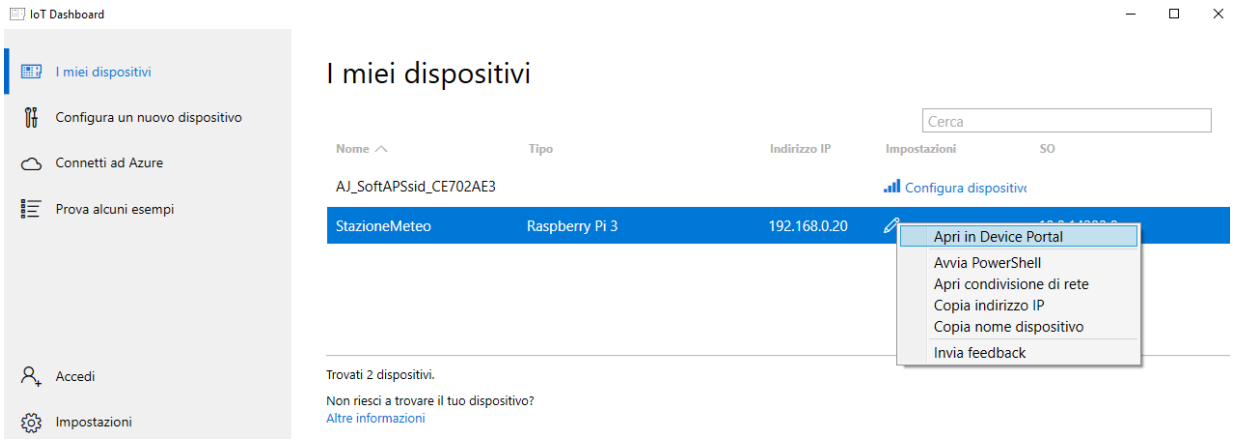


Fig.65

Automaticamente viene avviato il browser di default con la richiesta di autenticazione che prevede l'inserimento delle credenziali come account "administrator". Si veda Fig.66.

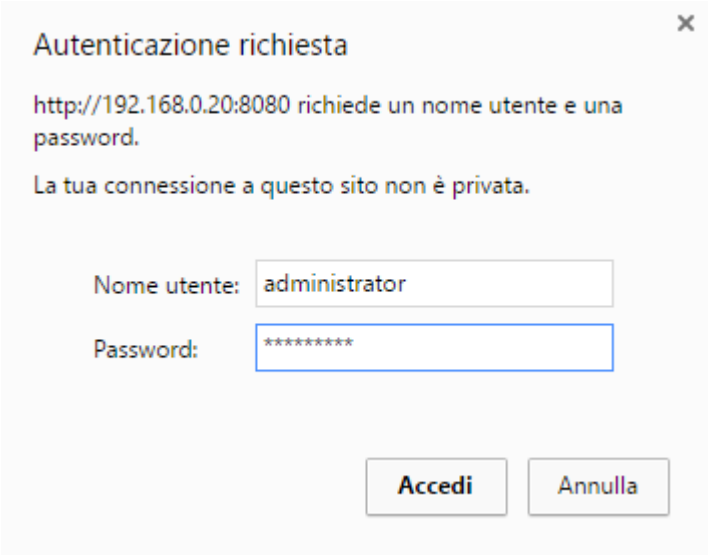


Fig.66

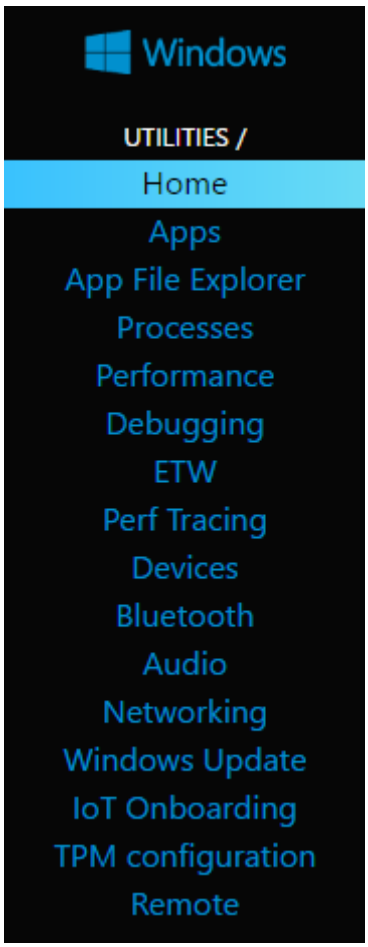


Fig.67

La pagina di default per l'amministrazione remota presenta nella sezione di sinistra il menù con tutte le utility necessarie alla configurazione del dispositivo in modo semplice ed immediato. La Fig.67 riporta la parte del solo menù mentre in Fig.68 sono indicate le informazioni base del Pi3, a partire in primis della versione del sistema operativo che risulta essere la 10.0.14393.0.

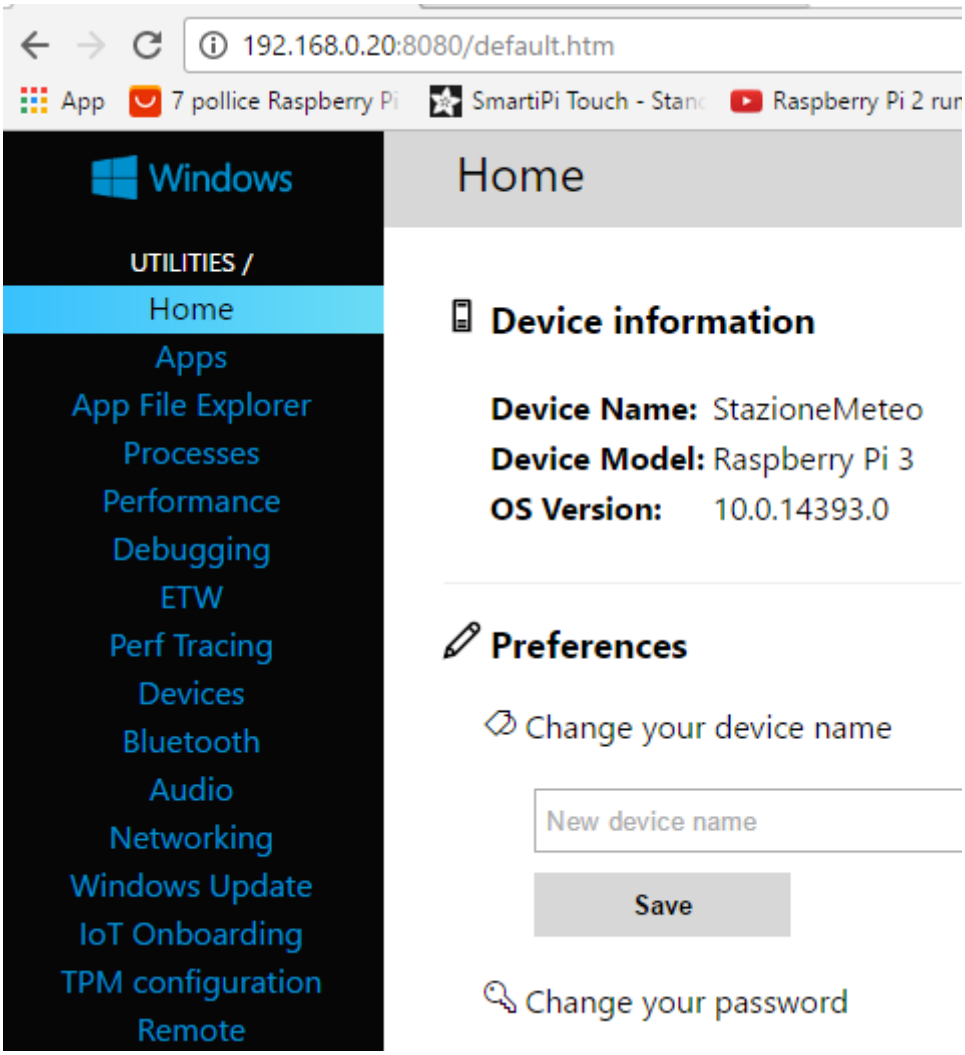


Fig.68

Premendo sulla voce "Networking" si apre la sezione con i parametri dello stack IPv4 e IPv6, nella quale è possibile impostare la configurazione delle reti WiFi 802.11x, anche se in questo progetto verrà utilizzata esclusivamente una connessione diretta FastEthernet 802.3u a 100Mbit/s. Il nome del device di rete è indicato nella parola chiave "Type". La Fig.69 riporta solo parte delle informazioni che si trovano in tale sezione.

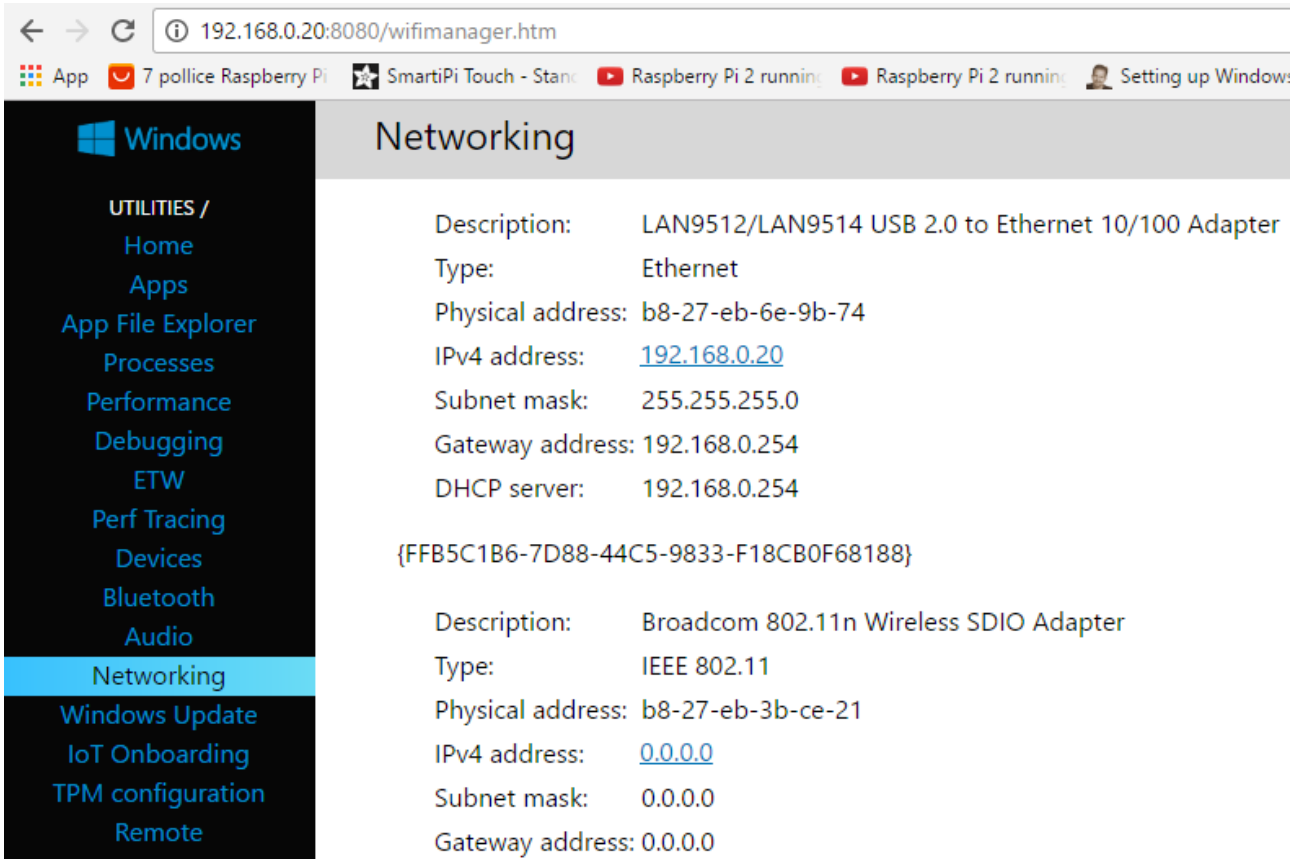


Fig.69

E' interessante la presenza della sezione dedicata all'aggiornamento della piattaforma tramite il classico Windows Update che procede al download delle patch del sistema operativo. La fase di download della dashboard aveva scaricato la Build 14393, ma tale sezione provvederà ad aggiornare tale costruzione variando il parametro dopo il punto, ossia ad esempio da 14393.0 a 13.393.100. Prima di inserire il device in un ambiente di produzione è saggio verificare la presenza di aggiornamenti e testare correttamente il funzionamento della propria app anche effettuando appositamente dei restart. La Fig.70 mostra la fase di ricerca e download dei pacchetti la quale può durare parecchi minuti, così come l'installazione degli aggiornamenti, quindi se dopo il riavvio la dashboard non rileva il Pi è perché il sistema al riavvio procede all'installazione delle patch, come accade con il sistema operativo versione Home. La durata di questa fase può arrivare anche a venti minuti, il tutto legato alla dimensione del download e al tipo di aggiornamento che comunque resta trasparente all'amministratore del Raspberry. Durante la fase di aggiornamento è interessante vedere le performance di utilizzo della CPU e della ram tramite la sezione "Performance" di Fig.71.

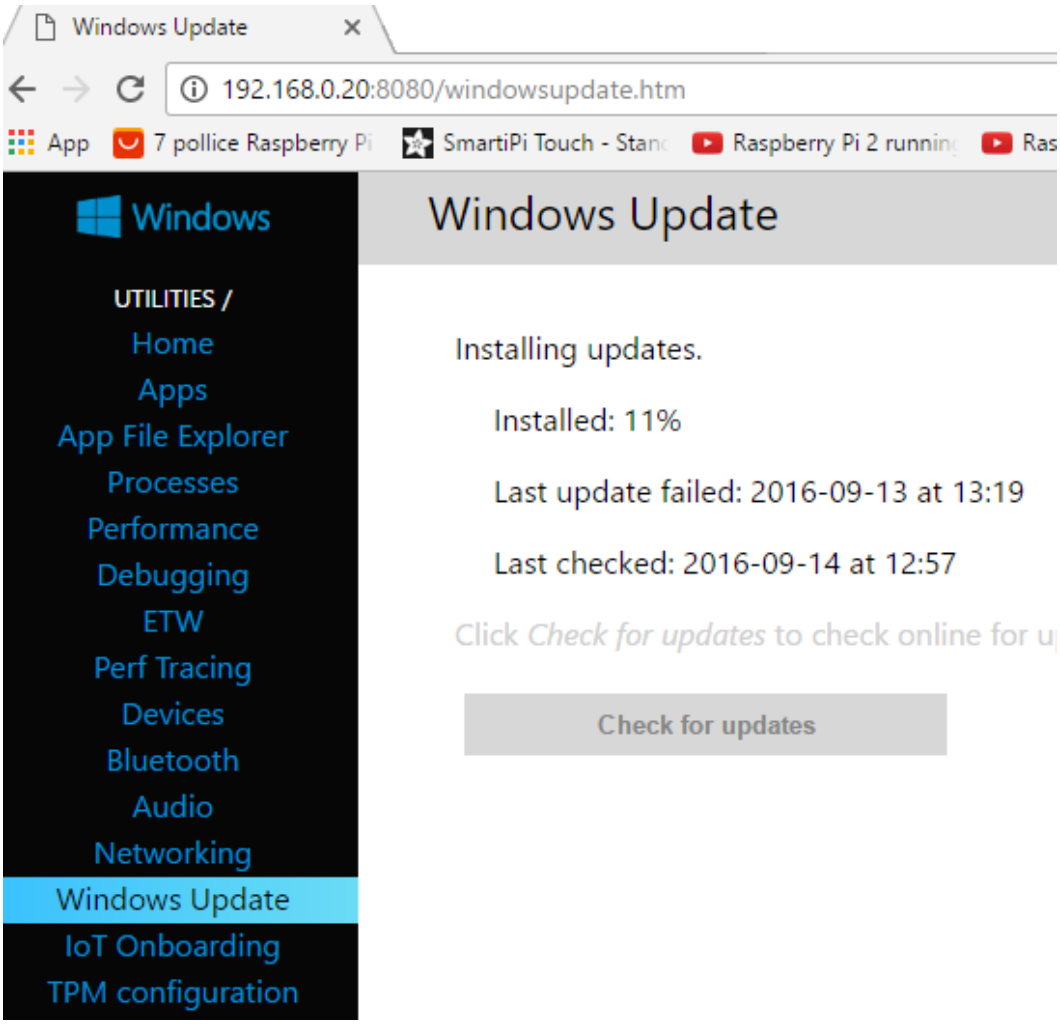


Fig.70

L'utilizzo medio della ram senza nessuna app installata è di circa 500MB.

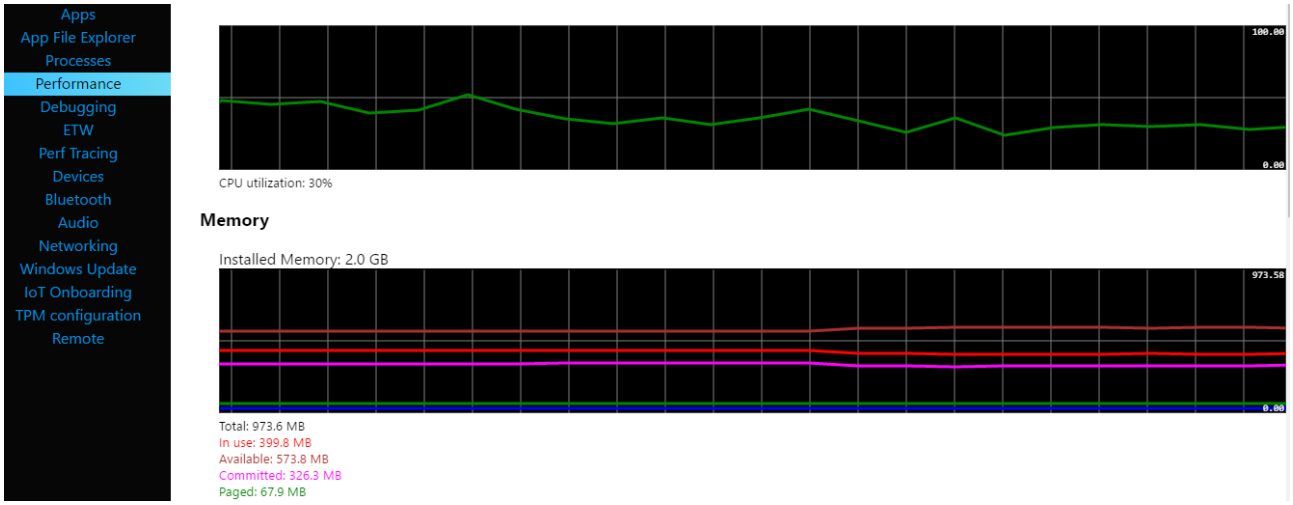


Fig.71

La configurazione dello stack IPv4 per default utilizza il DHCP, quindi in un ambiente di produzione conviene assegnare un IP statico al Pi così che possa essere sempre identificato dal medesimo indirizzo, nonostante sia possibile impostare delle "DHCP Reservation" con le quali assegnare in modo dinamico sempre lo stesso indirizzo IP. Per modificare lo stack IPv4 sulla scheda Ethernet (ma identico discorso anche per il WiFi) è necessario impiegare la "PowerShell" avviabile con privilegi di amministratore direttamente dalla dashboard, tramite il solito menù del tasto destro del mouse. La Fig.72 mostra l'avvio della "PowerShell" mentre la Fig.73 l'inserimento delle credenziali di "Administrator".

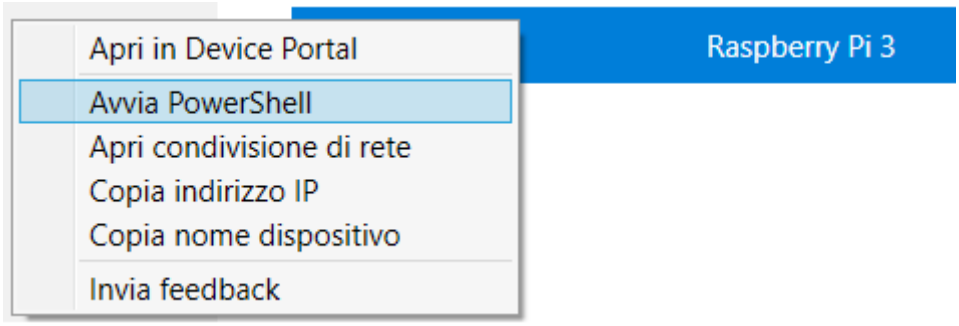


Fig.72

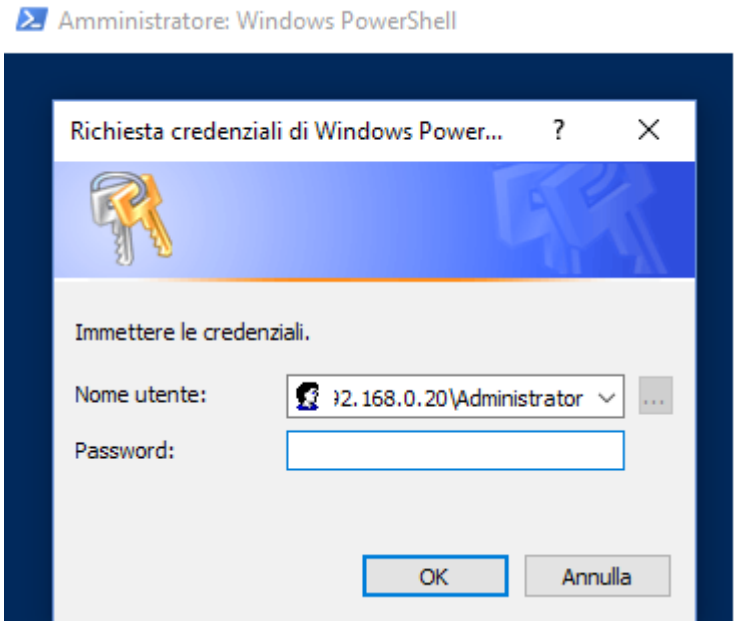


Fig.73

Prima di procedere all'impiego dei comandi, è bene ricordare il nome del device sul quale si vuole modificare lo stack IPv4, quindi ricordando la sezione di rete con il parametro "Type", è necessario utilizzare il nome "Ethernet" per invocare la modifica voluta. La Fig.74 riporta solo la parte del pannello relativa alla scheda di rete FastEthernet 802.3u.

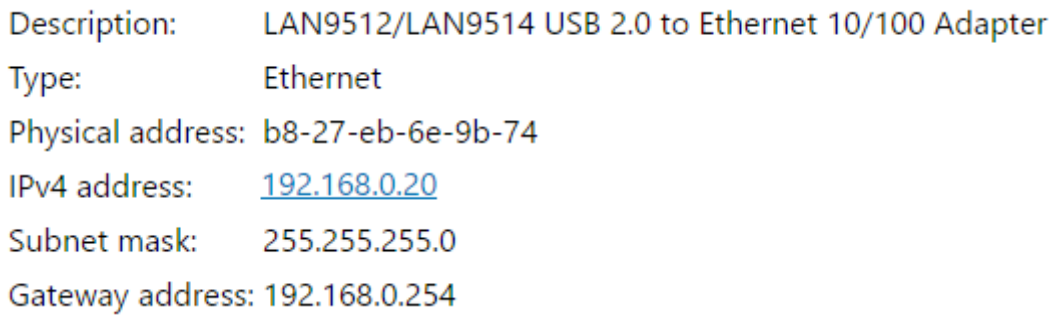


Fig.74

La scelta dell'indirizzo IP da utilizzare è "192.168.0.100/24", mentre per il default gateway e DNS si utilizza "192.168.0.254".
Il comando da utilizzare nella shell è quello di Fig.75 riportato sotto per semplicità di lettura.

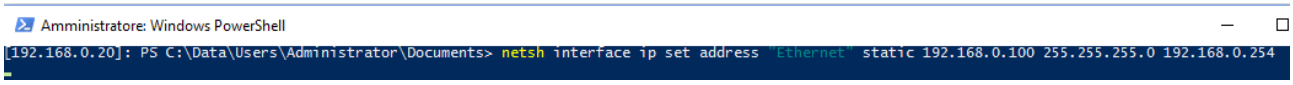


Fig.75

netsh interface ip set address "Ethernet" static 192.168.0.100 255.255.255.0 192.168.0.254

Una volta impartito il comando la shell si blocca visto la modifica dello stack, quindi è necessario chiudere e riaprire la "PowerShell". La dashboard rileverà immediatamente la modifica dell'indirizzo IP. La modifica del DNS è indicata in Fig.76 e riportata per comodità sotto.

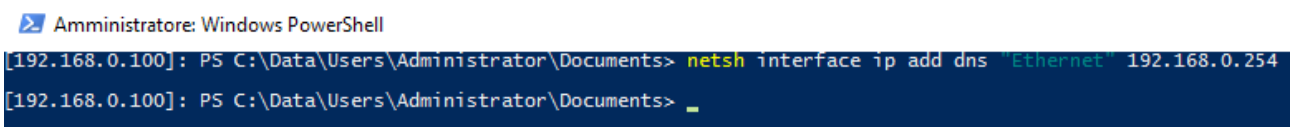


Fig.76

netsh interface ip add dns "Ethernet" 192.168.0.254

Il prompt dei comandi ritorna immediatamente visto che la modifica non interessa l'indirizzo IP utilizzato per la connessione.
Il risultato complessivo della modifica dei parametri dello stack è visibile in dashboard come da Fig.77.

Fig.77

La fase di configurazione base del Pi è terminata, conviene quindi procedere allo spegnimento del device qualora non serva accesso per scopi diversi da questa trattazione. Collegandosi remotamente via browser utilizzando la dashboard, come fatto in precedenza in Fig.65, procedere alla shutdown tramite la relativa voce del menù "Power" di Fig.78 che provoca lo spegnimento del Pi come da Fig.79.

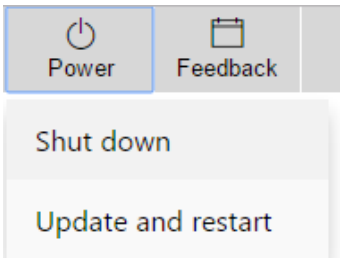


Fig.78

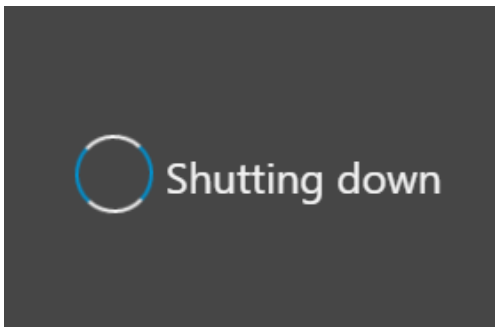


Fig.79

Capitolo 3

Testing scheda Pi 3 con C#

Questo capitolo è dedicato al testing hardware della GPIO tramite dei semplici programmi C# sviluppati con Visual Studio 2015.

3.1 *Visual Studio 2015 Enterprise*

Il software Visual Studio è un potente ambiente di sviluppo creato da Microsoft che permette la realizzazione, in modo semplice e robusto, di applicazioni di vario genere sfruttando molti linguaggi di programmazione basati sul framework .NET come C# e VB, ma al tempo stesso permette l'uso di codice non gestito C/C++.

3.1.1 Download ed installazione

La release corrente di Visual Studio è la 2015 offerta in alcune versioni, tra cui spicca quella di uso gratuito che è la "Community Edition", fino a quella professionale che è la "Enterprise" soggetta però a costi di licenza. In questo progetto verrà utilizzato la release "Enterprise", ma non cambia nulla in fase di realizzazione se si procede ad impiegare la versione "Community Edition". L'utente potrebbe anche utilizzare altri ambienti di sviluppo opensource reperibili sul mercato, visto che quello che conta è utilizzare il linguaggio C# e le librerie del framework .NET per lo sviluppo IoT. Per questioni di comodità, ma sopratutto di guida in linea molto dettagliata, si preferisce optare per l'ambiente di Microsoft. L'utilizzo della versione "Enterprise" prevede, come si vedrà nelle figure successive, il possesso di un numero seriale da inserire per l'attivazione del prodotto, previa registrazione di un account.

Il sito di riferimento è www.visualstudio.com come si vede dalla Fig.80.

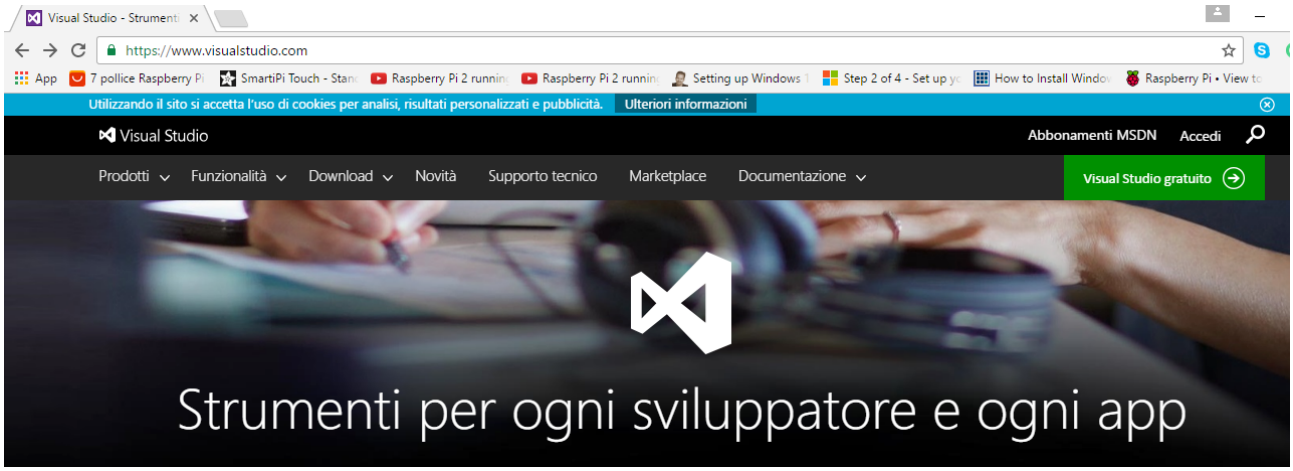


Fig.80

Nella sezione "Download" posizionarsi su "Download preferiti" e selezionare "Visual Studio Enterprise" come da Fig.81.

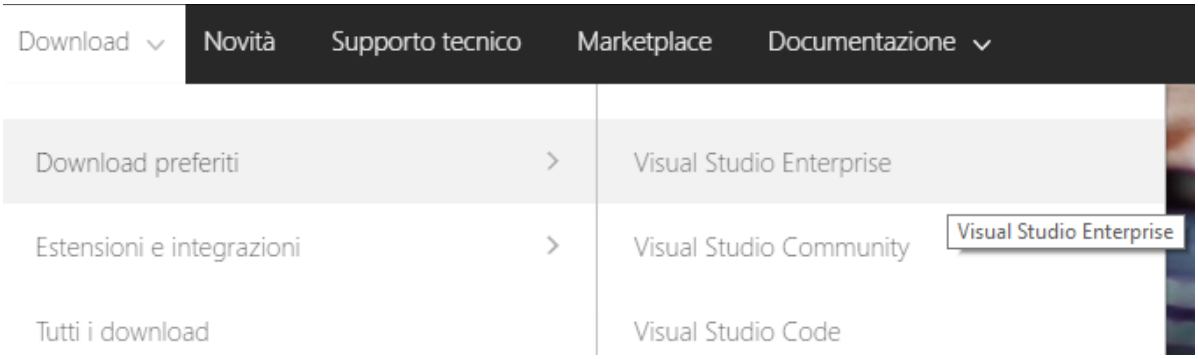


Fig.81

Il download procederà a scaricare un piccolo file, il quale una volta avviato, presenterà la tipologia di installazione che volete utilizzare, come si evince da Fig.82. La "Typical" garantisce un'allocazione di spazio pari a 7GB e la presenza dei linguaggi C# e VB. Qualora servisse il linguaggio C++ o avere la certezza di cosa viene installato, è meglio selezionare la configurazione "Custom" ed osservare le opzioni presenti di default, le quali poi possono venire modificate secondo le proprie esigenze.

Conviene lasciare la directory di installazione di default, così che eventuali aggiornamenti di qualsiasi natura non provochino errori difficilmente interpretabili, dovuti magari a bug nel media di aggiornamento.

Selezionare "Custom" e premere "Next" con il risultato di vedere le opzioni installate di default della "Typical", come da Fig.83.

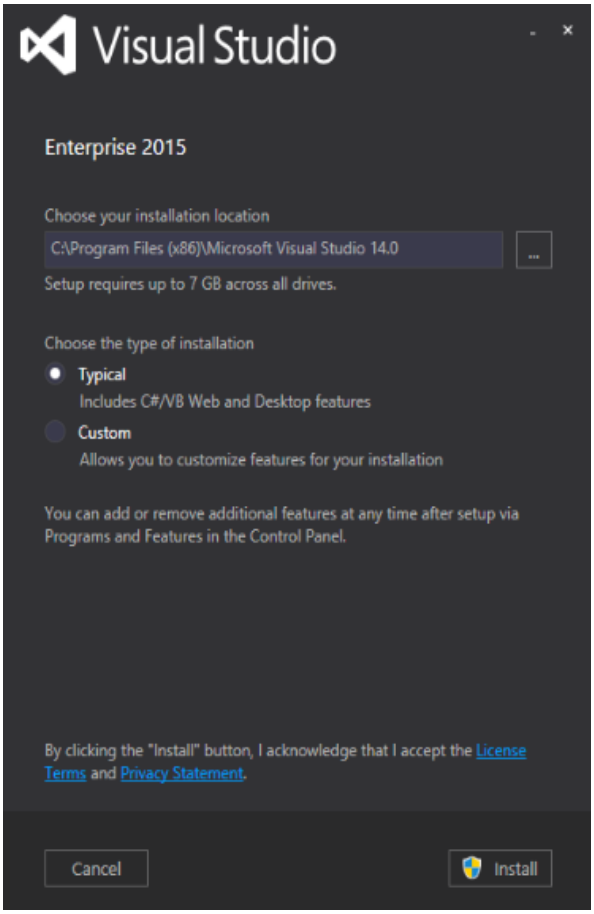


Fig.82

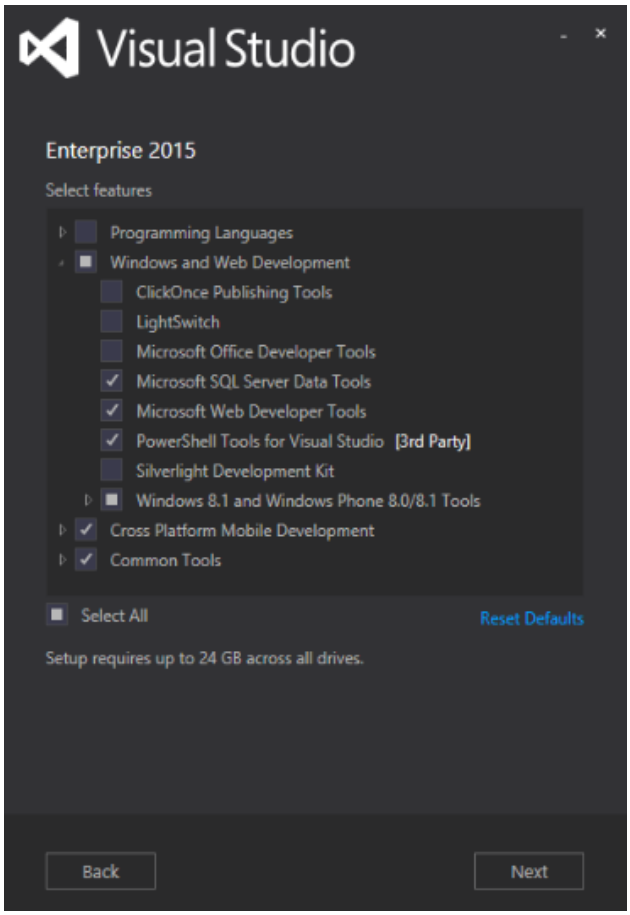


Fig.83

E' interessante osservare che un'installazione totale occupa al massimo 24GB, implementabile semplicemente con la "Select All". I linguaggi che verranno installati sono verificabili dalla sezione "Programming Languages". Procedere all'avvio automatico dei download dei pacchetti e alla loro installazione premendo il pulsante "Next", come da Fig.84. La procedura di download dipende molto dalla linea xDSL utilizzata, quindi è consigliato disporre almeno di una 10Mbit/s, così come l'installazione dei pacchetti è legata al tipo di calcolatore utilizzato in termini di microprocessore, memoria ram e memoria di massa secondaria. E' consigliato, al momento della stesura di questo documento, disporre di una CPU quad-core, almeno 8GB di ram ed un disco SSD (Solid State Drive) di taglio maggiore o superiore a 128GB. E' importante sottolineare che lo sviluppatore potrebbe avere l'esigenza di avviare sessioni multiple di Visual Studio, quindi la memoria ram impiegata potrebbe crescere vertiginosamente obbligando il sistema

operativo a lavorare molto di memoria virtuale, motivo per cui se possibile sarebbe il caso di avere a bordo 16GB.

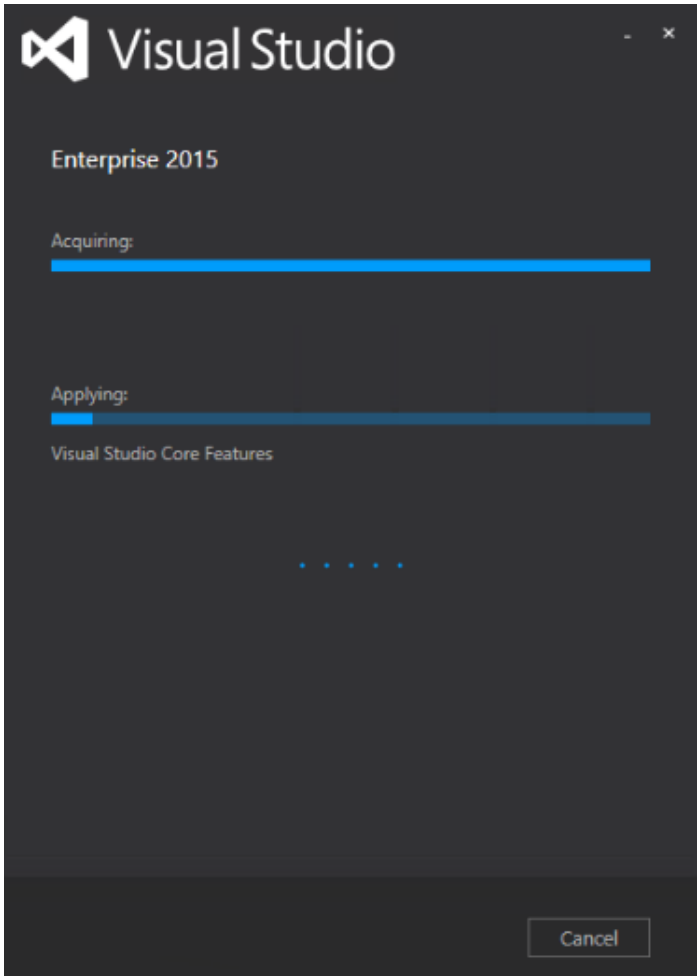


Fig.84

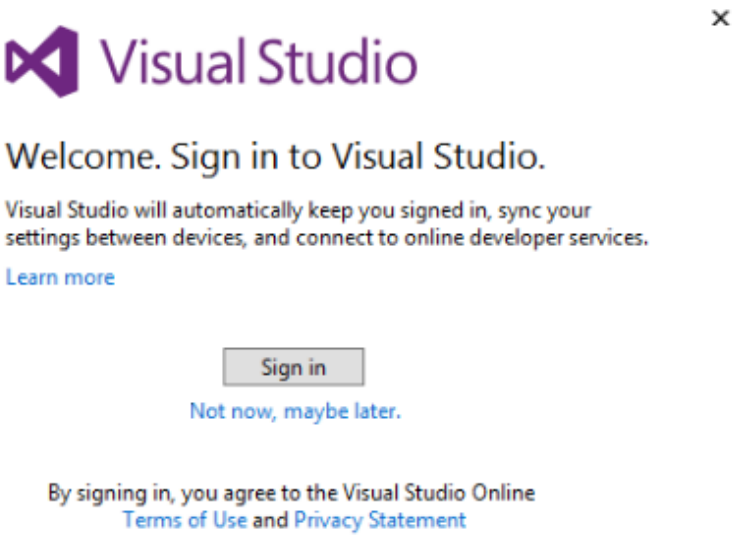


Fig.85

Al termine dell'installazione, come da Fig.85, procedere all'autenticazione con un'account Microsoft premendo il pulsante "Sign in", oppure rimandare questa operazione premendo il link "Not now, maybe later" nel caso non si posseggano tali credenziali.
Il primo avvio di Visual Studio è molto lungo ed è fortemente legato al tipo di calcolatore utilizzato, visto che si procede alla creazione della guida in linea, del profilo legato all'account locale e di altri parametri base. La Fig.86 mostra la schermata di attesa.

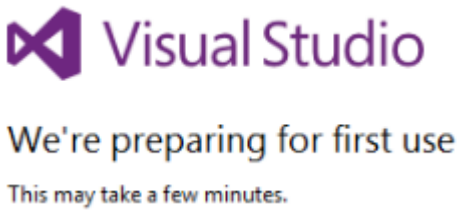


Fig.86

Una volta avviato l'ambiente di sviluppo vi sono 30 giorni prima di procedere alla registrazione del prodotto, che può essere fatta tramite la voce "Register Product" del menù "Help" come da Fig.87.

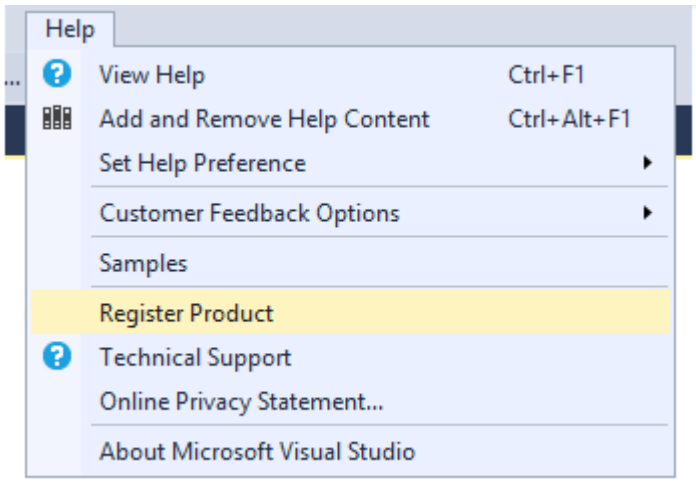


Fig.87

Successivamente effettuare la login qualora non sia stata fatta in precedenza ed inserire il numero seriale del prodotto tramite il link "License with a Product Key". Si veda la Fig.88.

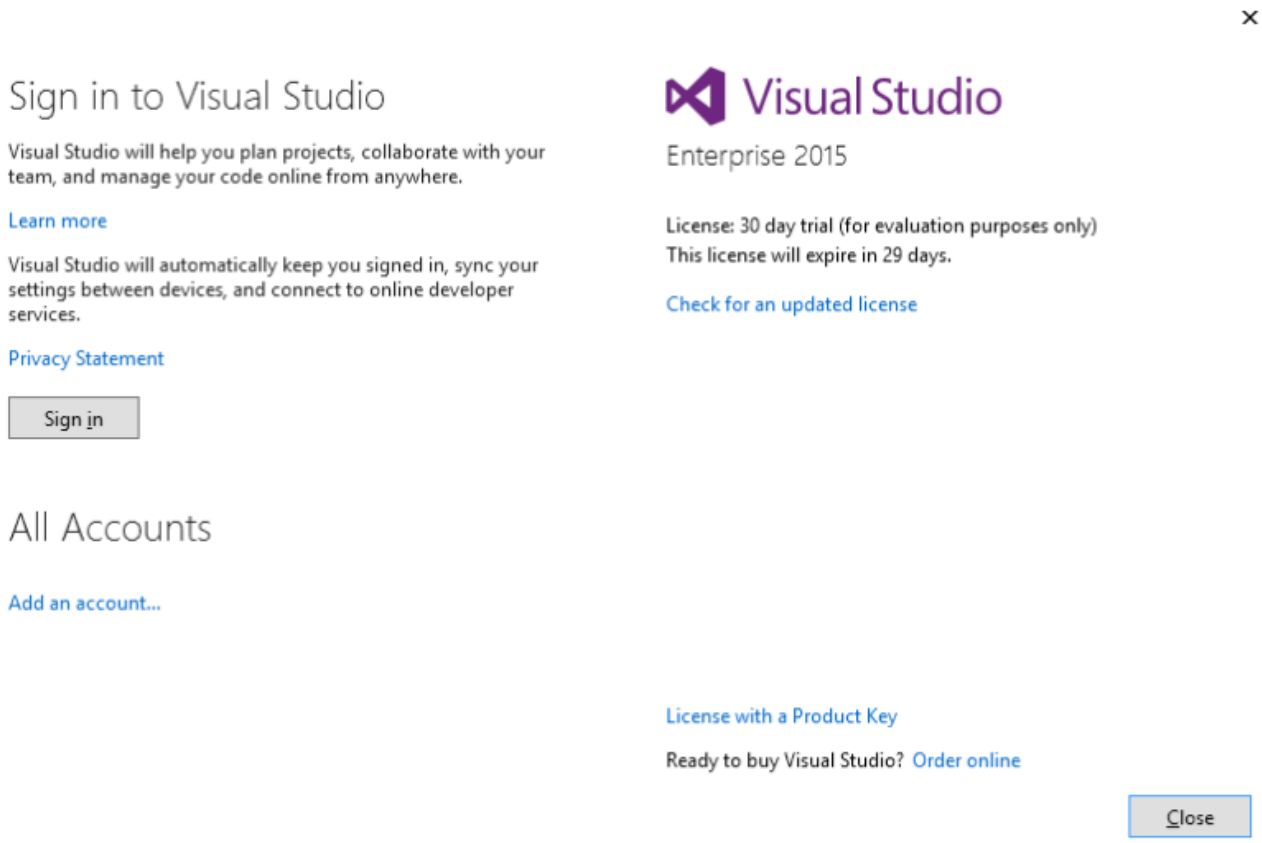


Fig.88

Inserire il "product key" e premere il pulsante "Apply" come da Fig.89. La registrazione di Visual Studio termina in Fig.90.

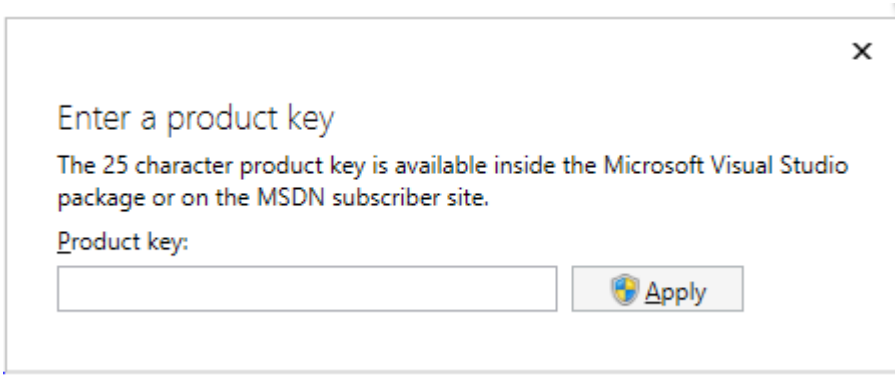


Fig.89

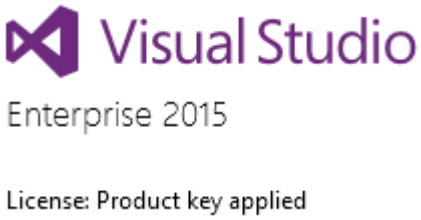


Fig.90

3.1.2 Windows 10 IoT Core Templates

Dopo l'installazione di Visual Studio 2015 è necessario installare il pacchetto "Windows 10 IoT Core Templates" che provvede ad inserire dei templates per la programmazione coi sistemi IoT tramite i linguaggi C#/VB/C++ e Javascript. Il link del pacchetto è https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ad-82a5-a88ac4b01dec come da Fig.91 ed è scaricabile tramite il pulsante "Download".

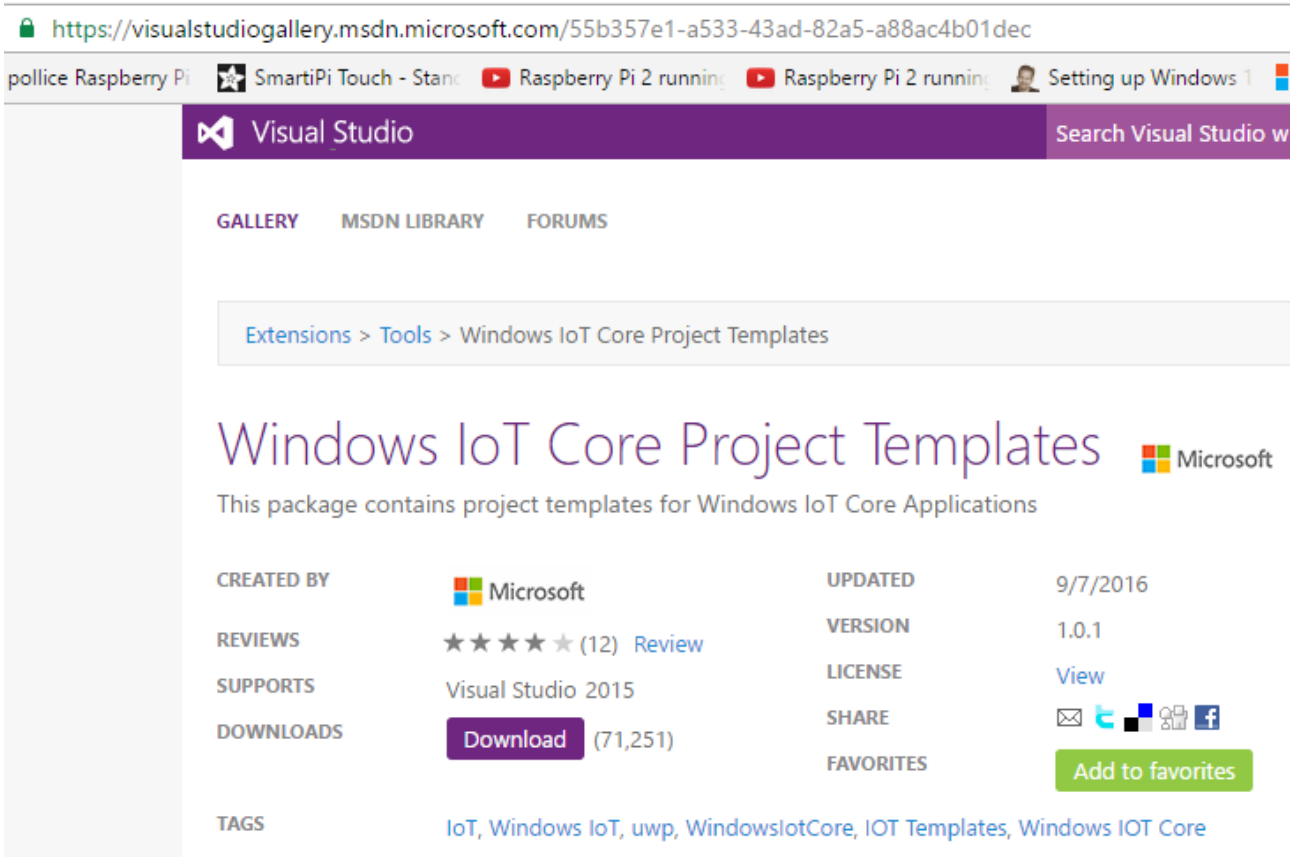


Fig.91

L'esecuzione del pacchetto avvia il media di installazione e la richiesta di esecuzione con i privilegi di amministratore. La Fig.92, Fig.93 e Fig.94 mostrano i semplici ed immediati steps di installazione (le figure si riferiscono alla versione Community, ma non cambia nulla rispetto alla Enterprise).

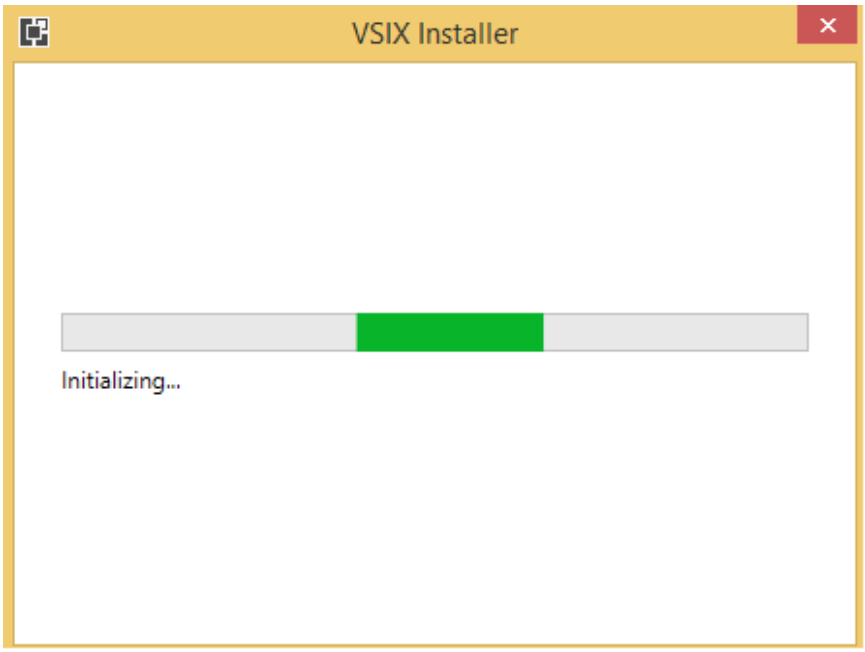


Fig.92

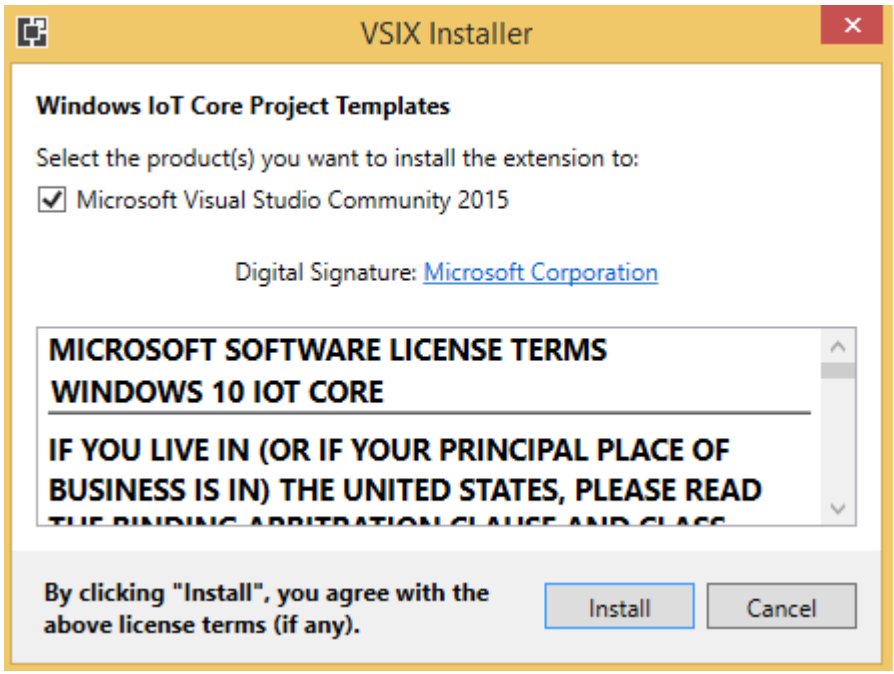


Fig.93

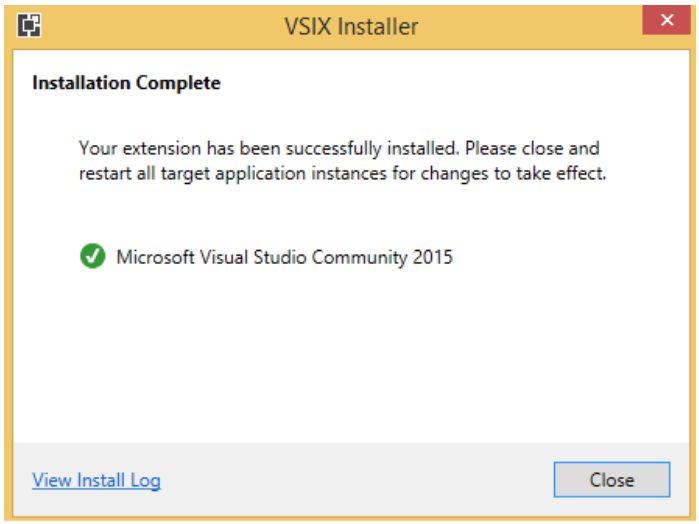


Fig.94

3.1.3 Visual Studio 2015 Update 3

Il passo finale deve comprendere l'applicazione degli aggiornamenti del Visual Studio 2015, anche se durante l'installazione effettuata in precedenza, il media si scarica automaticamente sia il pacchetto "Update 3", che contiene un pool di aggiornamenti cospicui, sia eventuali altre patch che risolvono problemi vari. In Fig.95.

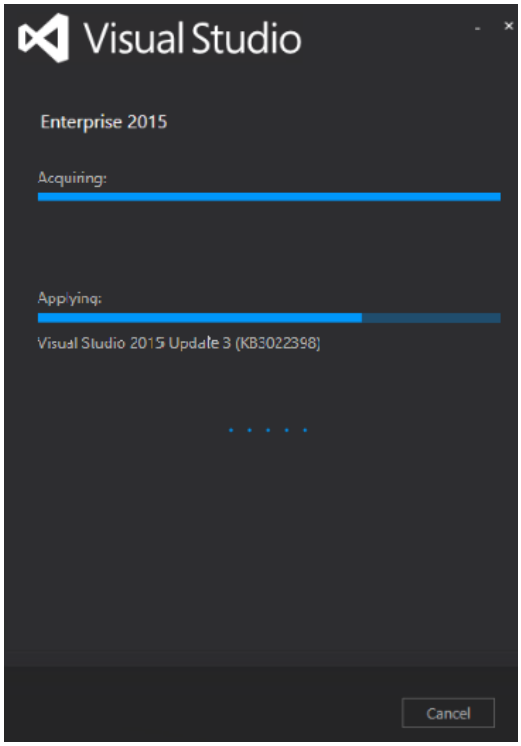


Fig.95

Qualora l'utente abbia già installato una versione di Visual Studio 2015 subito dopo il rilascio, dovrà applicare il pacchetto "Update 3" scaricandolo direttamente dal sito o utilizzando il packet manager integrato nell'ambiente. La prima cosa è verificare la release corrente tramite la voce di "About Microsoft Visual Studio" del menù "Help" di Fig.96.

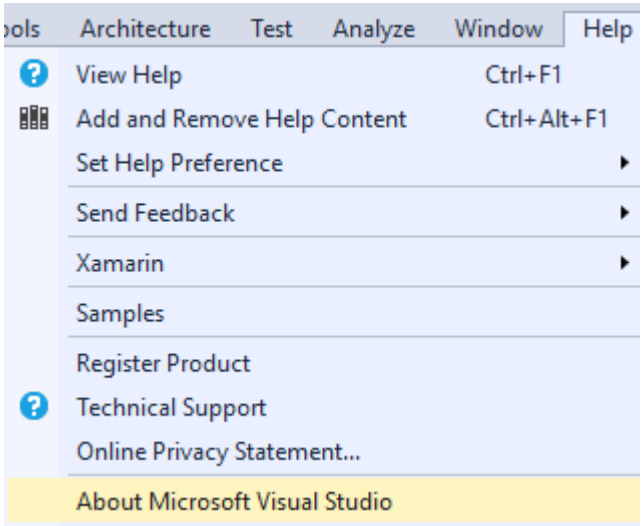


Fig.96

La release installata è la "Update 1" di Fig.97 che necessità quindi di venire aggiornata.

Fig.97

La prima strada per l'aggiornamento è scaricare il pacchetto "Update 3" direttamente dal sito di Microsoft https://www.visualstudio.com/news/releasenotes/vs2015-update3-vs come si evince dalla Fig.98, avviando poi il media di installazione, che ricalca quello utilizzato dell'ambiente Visual Studio, selezionare/deselezionare eventualmente le parti interessate come in Fig.99 e procedere all'aggiornamento con il pulsante "Update".

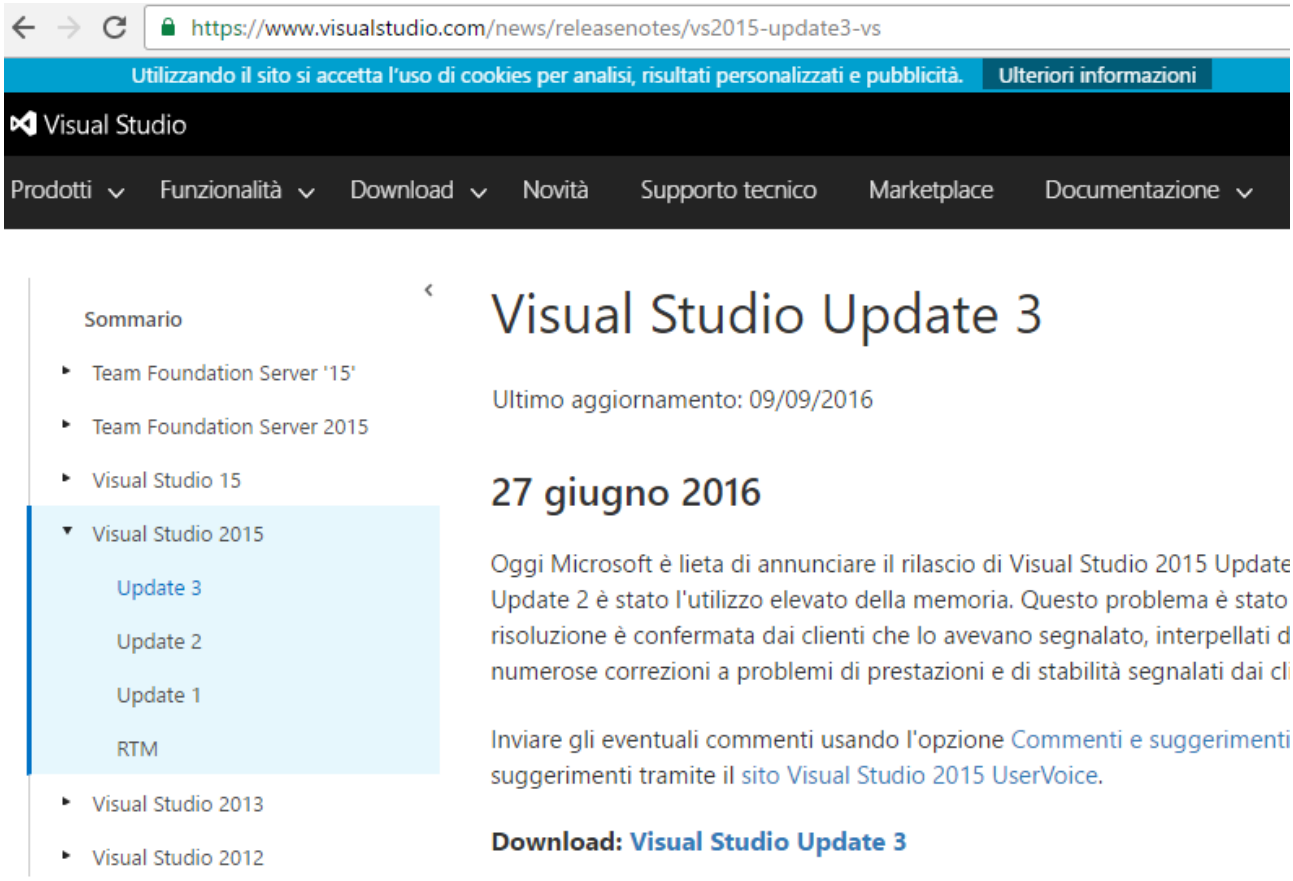

https://www.visualstudio.com/news/releasenotes/vs2015-update3-vs
Utilizzando il sito si accetta l'uso di cookies per analisi, risultati personalizzati e pubblicità.
Ulteriori informazioni
Visual Studio
Prodotti
Funzionalità
Download
Novità
Supporto tecnico
Marketplace
Documentazione
Sommario
Team Foundation Server '15'
Team Foundation Server 2015
Visual Studio 15
Visual Studio 2015
Update 3
Update 2
Update 1
RTM
Visual Studio 2013
Visual Studio 2012
Visual Studio Update 3
Ultimo aggiornamento: 09/09/2016
27 giugno 2016
Oggi Microsoft è lieta di annunciare il rilascio di Visual Studio 2015 Update
Update 2 è stato l'utilizzo elevato della memoria. Questo problema è stato
risoluzione è confermata dai clienti che lo avevano segnalato, interpellati d
numerose correzioni a problemi di prestazioni e di stabilità segnalati dai cl
Inviare gli eventuali commenti usando l'opzione Commenti e suggerimenti
suggerimenti tramite il sito Visual Studio 2015 UserVoice.
Download: Visual Studio Update 3

Fig.98

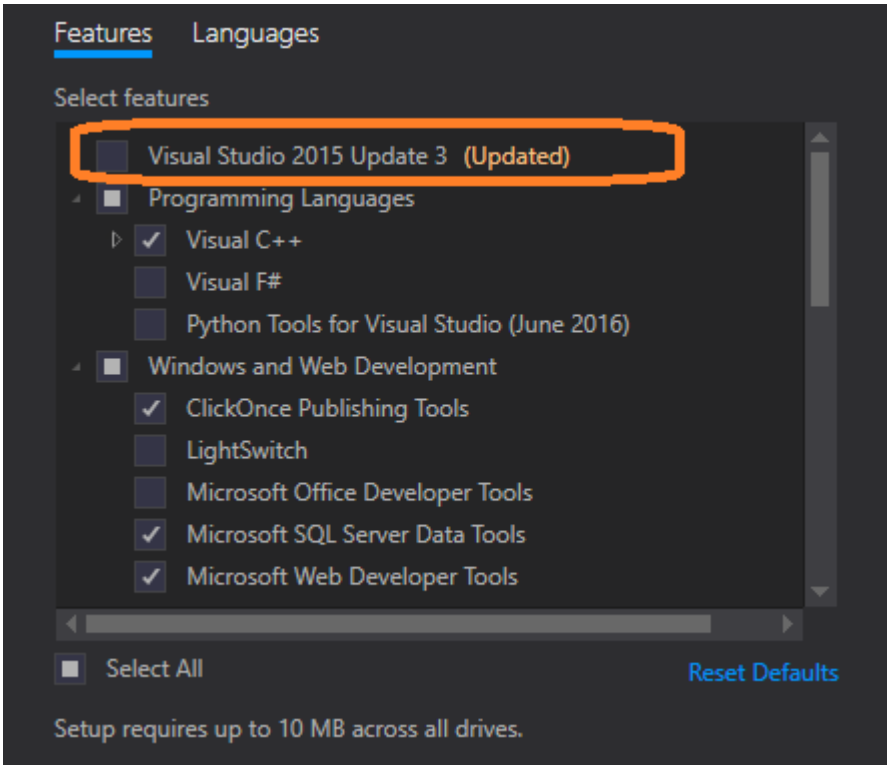

Features
Languages
Select features
Visual Studio 2015 Update 3 (Updated)
Programming Languages
Visual C++
Visual F#
Python Tools for Visual Studio (June 2016)
Windows and Web Development
ClickOnce Publishing Tools
LightSwitch
Microsoft Office Developer Tools
Microsoft SQL Server Data Tools
Microsoft Web Developer Tools
Select All
Reset Defaults
Setup requires up to 10 MB across all drives.

Fig.99

La seconda strada è utilizzare il packet manager integrato nel Visual Studio, il quale elenca tutti i possibili update e patch rilasciate fino a quel momento da Microsoft e non presenti sul sistema. Per utilizzare il packet manager basta cliccare sull'icona a forma di bandiera presente in alto a destra, come da Fig.100, nella quale si notificano 4 update.

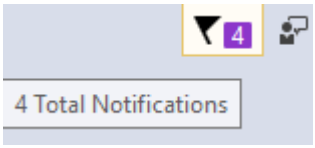


Fig.100

L'elenco degli update viene presentato come in Fig.101, quindi basta premere sul pulsante "Update" e scaricare il media di installazione che procederà all'aggiornamento nello stesso modo descritto in precedenza. Stesso discorso vale per gli altri tipi di aggiornamenti.

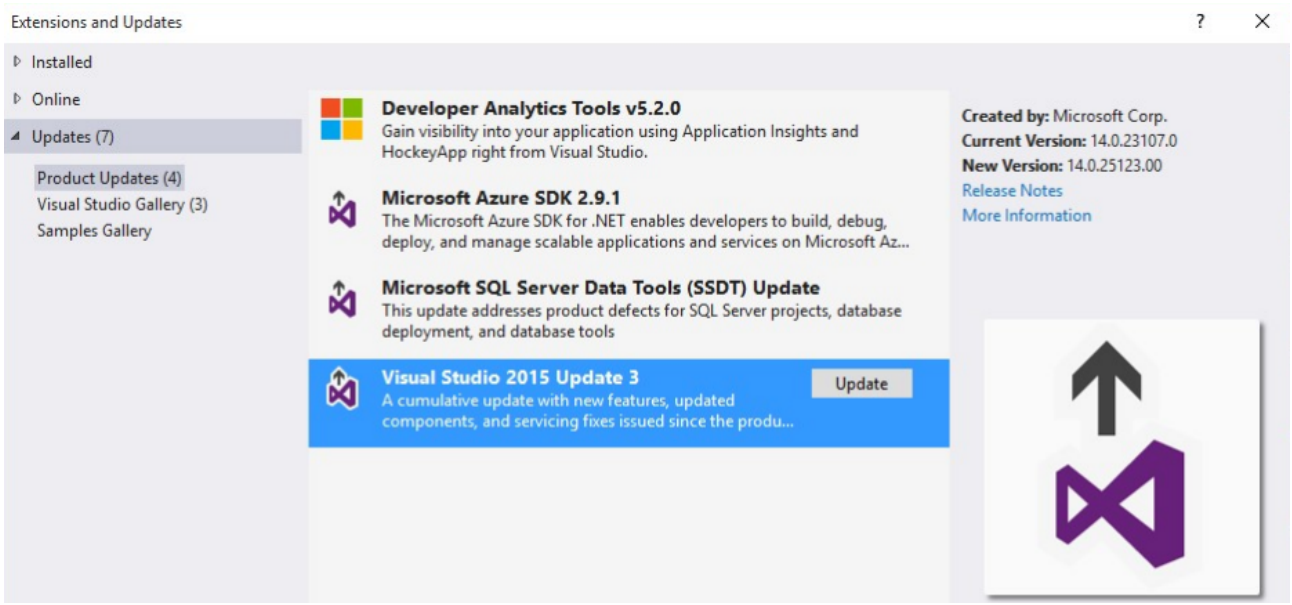


Fig.101

3.2 *Lampeggio di un led e pressione pulsante*

Il testing hardware di una scheda è un aspetto molto importante per garantire scalabilità di progetto ed evitare problemi di stalli improvvisi del software molto difficili da monitorare e capire. Il più semplice dei test che è possibile fare è verificare il corretto funzionamento di tutti i pin in configurazione entrata/uscita della GPIO, periferica cardine per l'interfacciamento. Il test può semplicemente essere realizzato tramite l'accensione e spegnimento di un diodo led col fine di verificare l'uscita del pin GPIO o leggere lo stato di pressione di un pulsante per verificare l'entrata del pin GPIO.

3.2.1 C#

La scrittura del codice di testing e del progetto stazione meteo viene fatta con il linguaggio gestito C#, il quale oltre che essere orientato agli oggetti, è multi piattaforma grazie alla presenza del CLR (Common Language Runtime) che funge da layer tra l'applicazione .NET ed il sistema operativo. Il CLR sfrutta la presenza del CIL (Common Intermediate Language) così da creare del codice che sia portabile su altri sistemi operativi, così che poi il CLR installato in loco funga da macchina virtuale per l'esecuzione dell'applicazione. Il risultato prodotto del CIL è un byte code, come accade nel mondo Java. La Fig.102 riassume quanto esposto.

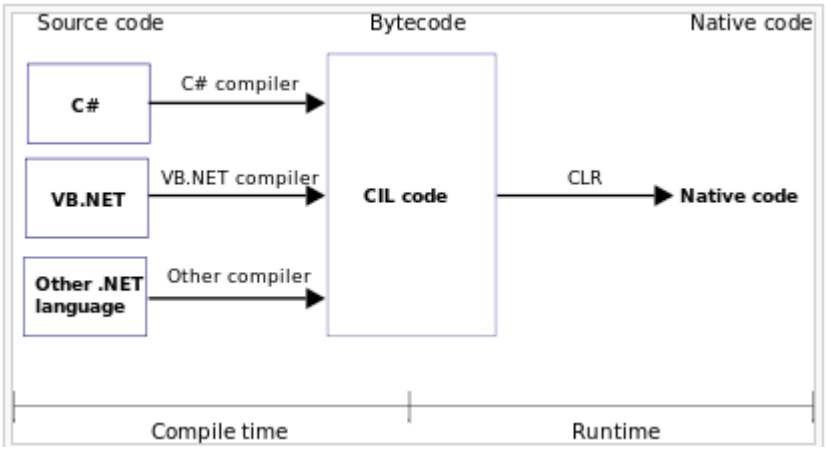


Fig.102

E' importante sottolineare che la presenza del CLR offre moltissimi vantaggi, ma anche alcune problematiche proprio nell'ambito dei sistemi embedded. Il primo grande vantaggio è appunto la portabilità del codice, visto che il framework .NET a cui si appoggia C#/VB, è

presente anche in Linux con il framework Mono, compatibile con .NET. Questo permette di scrivere del codice in ambedue i sistemi. Un altro vantaggio è la gestione della memoria HEAP, che viene affidata completamente al CLR, mentre nei linguaggi come C/C++ è il programmatore che deve gestire lo HEAP tramite le chiamate malloc/new per allocare spazio e free/delete per deallocare. Questo onere è fonte di guai, visto che l'uso dei puntatori in C/C++ è ambiguo, nonostante il C++ 2011 abbia introdotto gli smart pointer per risolvere i problemi di memory leakage. Il C# invece è un linguaggio gestito, ossia è il CLR che si occupa del ciclo di vita degli oggetti, provvedendo alla loro rimozione dello HEAP quando non sono più raggiungibili. Questa caratteristica permette al programmatore di dimenticarsi della gestione della memoria e quindi dei puntatori. La Fig.103 mostra un esempio di allocazione nello HEAP di una stringa, per la quale il flag di raggiungibilità è settato a "true", di conseguenza il CLR non distruggerà l'oggetto.

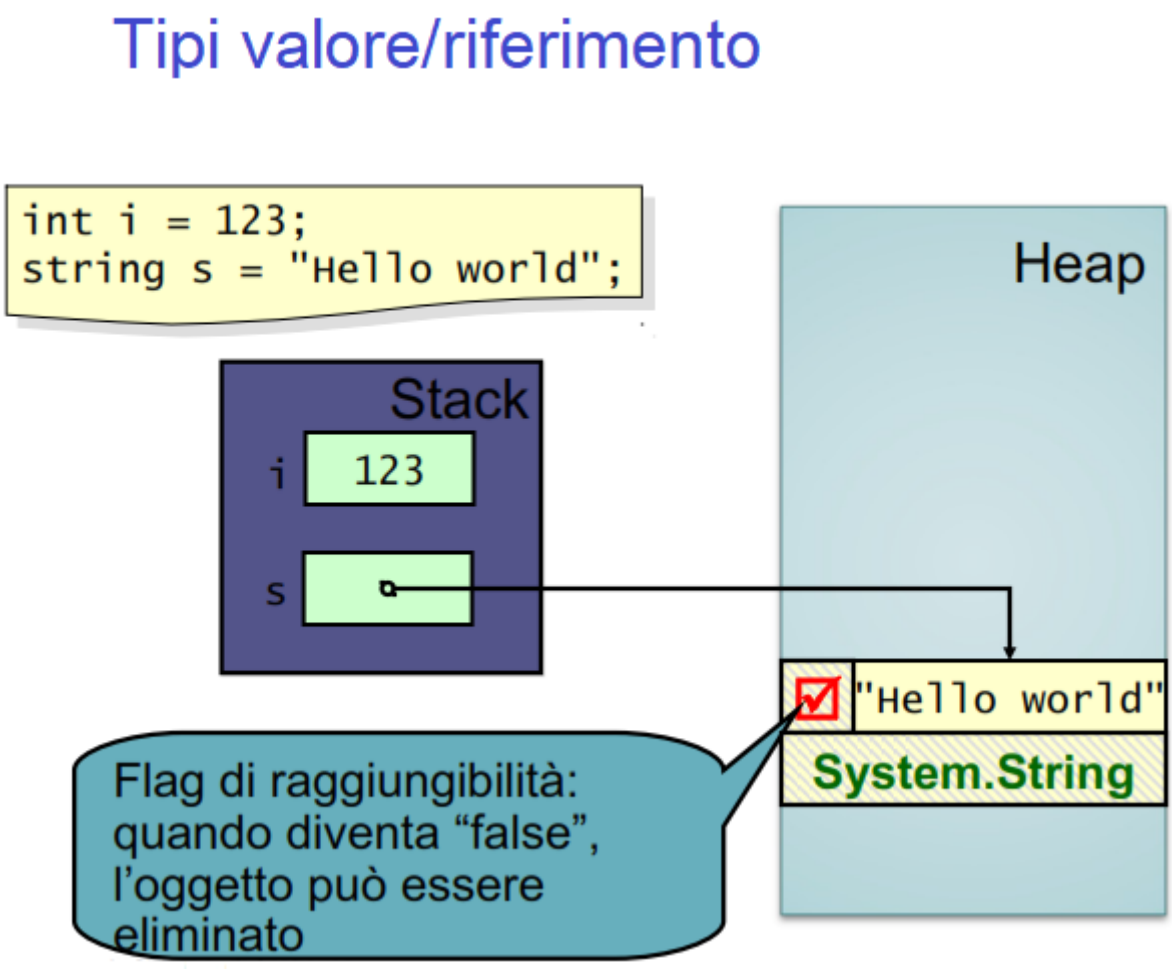


Fig.103

Lo svantaggio che si paga nell'impiego di un linguaggio gestito come C#/VB è la presenza del meccanismo di "Full Cycle Garbage Collector", che serve proprio per la pulizia dello HEAP, ma che obbliga l'applicazione al completo "freezing", ossia l'applicazione si blocca e qualsiasi richiesta va in timeout perché il Garbage Collector deve poter operare spostamenti nello HEAP per poi riassegnare i corretti indirizzi ai puntatori. Tale operazione in un ambiente desktop può durare anche un paio di secondi, anche se la presenza della memoria virtuale tende a ridurre notevolmente le volte in cui si verifica il "Full Cycle". Il problema è molto più serio in un sistema embedded, il quale non dispone di memoria virtuale ed il quantitativo a bordo di memoria ram è alquanto limitato. Bloccare l'applicazione che gestisce l'ABS di un'autovettura può causare danni irreparabili. In tali contesti sono più indicati linguaggi Real Time come il C/C++, dove con il termine Real Time non si intende maggiore velocità, ma esclusivamente la presenza di un limite di tempo superiore entro il quale il codice deve venire eseguito. Tale limite non potrà mai essere garantito in un ambiente gestito.

3.2.2 Circuito elettrico per l'accensione del led con fritzing

La parte di circuiteria elettrica per l'interconnessione di un diodo led è abbastanza elementare, ma può cambiare il valore logico, e quindi elettrico, della porta in base a come si vuole realizzare questa prima e banale esperienza, che in ogni caso porta ad avere una UWP completa e perfettamente funzionante. Per comodità si preferisce utilizzare due diodi led, uno rosso ed uno verde così da analizzare questi due approcci che portano al medesimo risultato. Prima di addentrarci nella scrittura della UWP, si deve progettare il circuito elettrico, quindi per tale operazione è molto comodo utilizzare un software opensource chiamato fritzing reperibile sul sito www.fritzing.org. Il pacchetto è multi piattaforma, come si evince dalla Fig.104, quindi basta scaricarlo per poi avviare l'applicazione tramite il file eseguibile "fritzing.exe". La creazione del progetto avviene tramite il menù "File" opzione "New", Fig.105, che provvede ad inserire nello spazio di lavoro una bread board sulla quale posizionare tutti gli elementi circuitali necessari per il progetto. Prima di iniziare la realizzazione del circuito, è necessario salvare il file fritzing assegnando, ad esempio, il nome "AccensioneLed" tramite l'opzione "Save as" del menù "File". Il file utilizza l'estensione ".fzz".

Fig.104

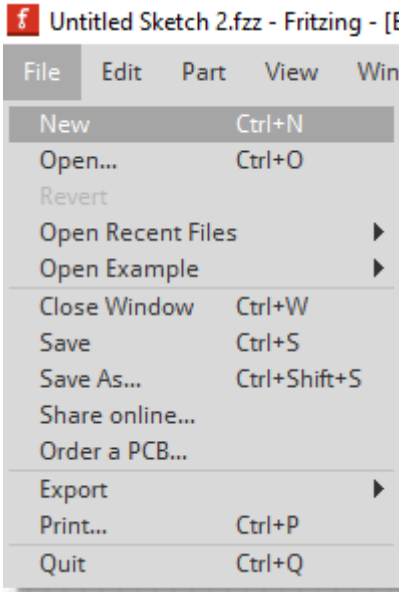


Fig.105

In Fig.106 riporta direttamente la realizzazione finale dei due circuiti per l'accensione rispettiva del diodo led rosso e verde.

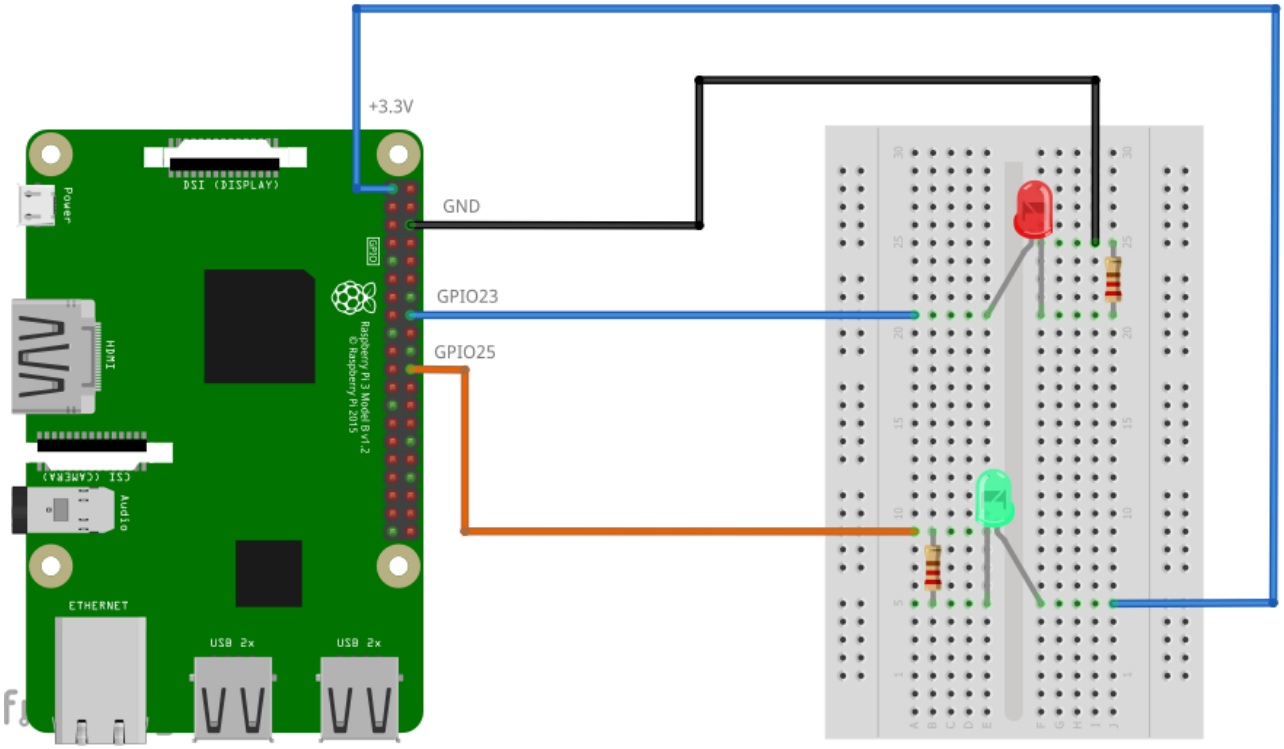


Fig.106

Il pin GPIO23 è connesso, tramite il cavo blu, all'anodo (+) del diodo led rosso, al quale in serie sul catodo (-) viene posto un resistore da 56Ω, il quale chiude la maglia a massa sul pin GND numero 3 della GPIO. Il led si accenderà se e solo se sul GPIO23 verrà riportata in uscita una tensione alta pari a +3.3V, in caso contrario non scorrendo corrente il led resta spento. Nel secondo circuito l'anodo (+) del diodo led verde è connesso, tramite un altro cavo blu, direttamente al potenziale alto di +3.3V sul pin numero 1 +3.3V PWR del GPIO, mentre il resistore, posto in serie al catodo (-) del led, viene connesso tramite il cavo arancione al pin GPIO25. Il led si accenderà solo se vi è il passaggio di corrente da un potenziale alto ad uno più basso, quindi il pin GPIO25 dovrà simulare la massa ponendo in uscita un valore di tensione basso.

Per entrambi i circuiti è sempre il Raspberry che pilota i pin GPIO, quindi la modalità di configurazione sia del GPIO23 che del GPIO25 è in uscita.

GPIO23 (diodo led rosso)	GPIO25 (diodo led verde)
pin configurato in uscita	pin configurato in uscita
tensione alta per accendere il led (High)	tensione bassa per accendere il led (Low)

Tab.7

Il dimensionamento del resistore è banale, considerando una tensione di alimentazione di +3.3V, una caduta di tensione di circa +2V ai capi del diodo led e una corrente massima di circolazione nella maglia pari a 8mA. I calcoli sottostanti mostrano il dimensionamento di una resistenza di circa 162Ω, anche se il valore commerciale prossimo è 150Ω oppure 180Ω. Si opta per un resistore di 150Ω da 1/4 di watt. In Fig.107 lo schematic del circuito.

$$V_{AL} = V_{LED} + V_R \Rightarrow V_{AL} = V_{LED} + R * I$$

$$\boxed{R = \frac{V_{AL} - V_{LED}}{I}}$$

$$R = \frac{5V - 3.3V}{8mA} = 162\Omega$$

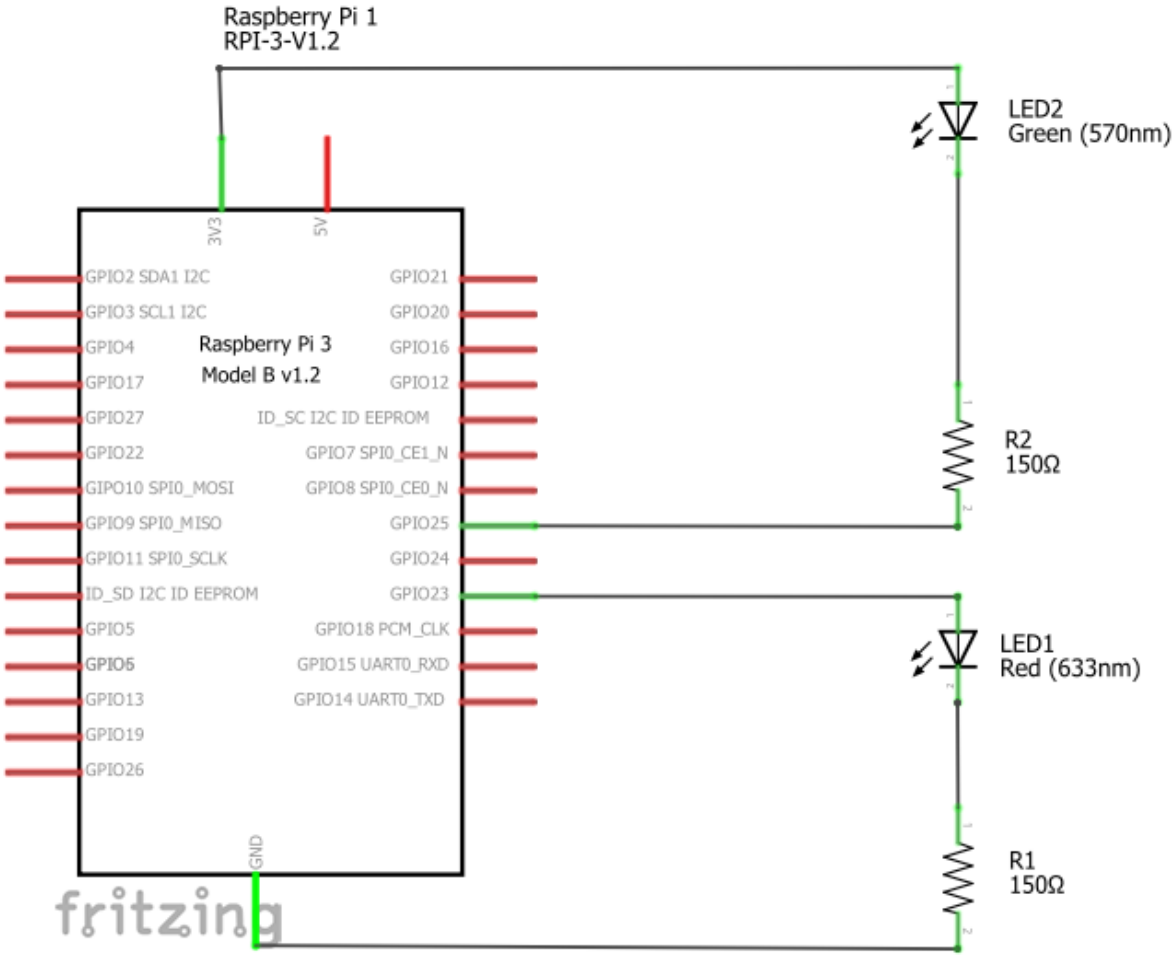


Fig.107

Il montaggio viene fatto seguendo fedelmente la Fig.106 e Fig.107, anche in termine di colore dei file così che sia più semplice il montaggio. In Fig.108 si vede l'inserimento dei cavi dei due circuiti nella GPIO, mentre nella Fig.109 e Fig.110 si vedono i due circuiti montati e funzionanti quando viene avviata la UWP sul Raspberry Pi3, parte che ancora non è stata trattata.

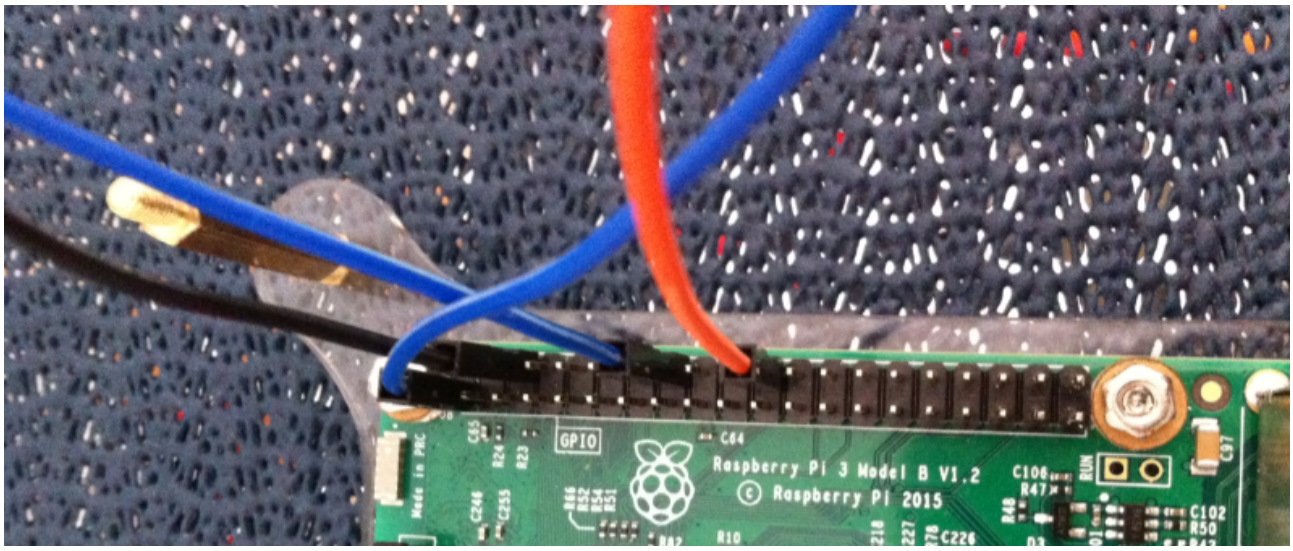


Fig.108

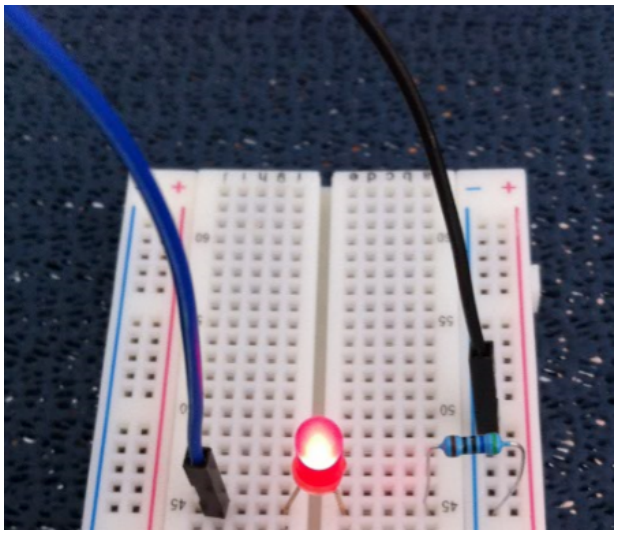


Fig.109

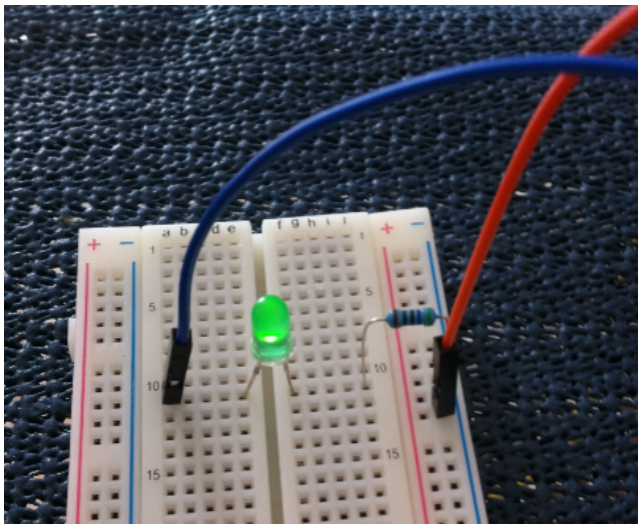


Fig.110

La foto della bread board completa è quella di Fig.111, mentre l'esecuzione della UWP su un monitor 48" è quella di Fig.112. Il design della UWP è composto da tre semplici textbox, come si vedrà nel paragrafo dedicato alla realizzazione del codice.

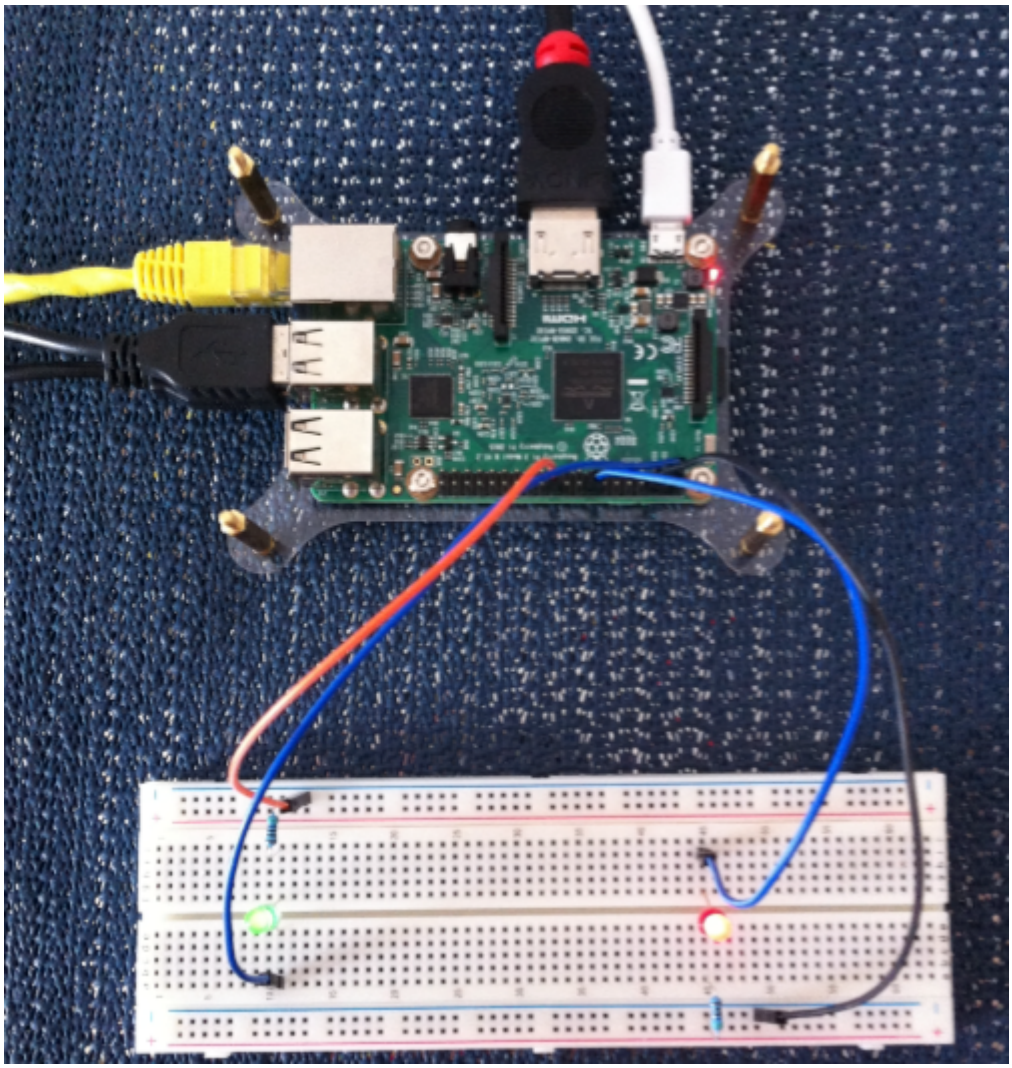


Fig.111

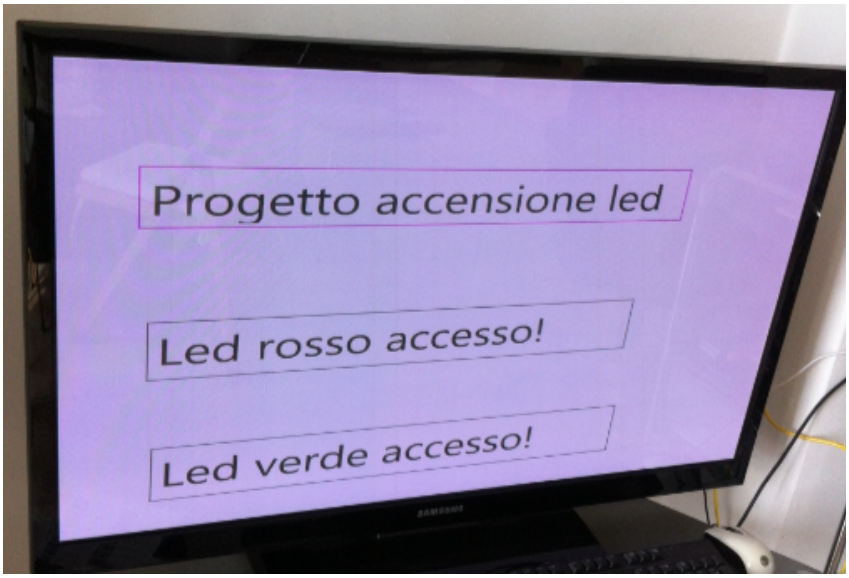


Fig.112

3.2.3 UWP

UWP(Universal Windows Platform) è un'architettura introdotta da Microsoft con Windows 10 per creare applicazioni universali capaci di funzionare su un insieme eterogeneo di dispositivi, come tablet, smartphone, TV, Xbox e device IoT i quali hanno come differenza la dimensione dello schermo o, nei device IoT, la totale assenza di un display. La Fig.113 mostra la scalabilità del sistema operativo Windows 10 su vari device.

Fig.113

Il core principale di UWP è un insieme di API universali le quali sono compatibili su tutti i device, quindi la realizzazione di una UWP garantisce il funzionamento su molteplici famiglie di dispositivi. Questo core principale definisce una famiglia universale di device dalla quale è possibile creare delle specializzazioni nel momento in cui l'utente decide di sviluppare per una determinata famiglia di device. La Fig.114 chiarisce questo aspetto.

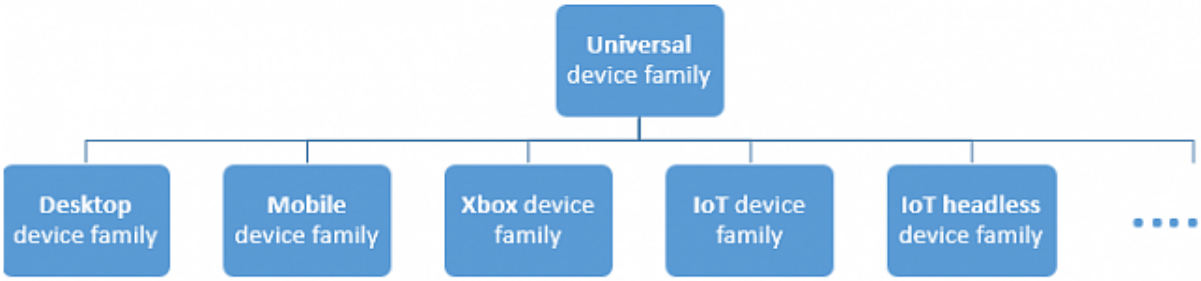


Fig.114

Per specializzare la UWP ad una data famiglia, come ad esempio la "Desktop device family" di Fig.114, basta utilizzare l'apposito SDK che contiene delle specifiche API per tali dispositivi. La Fig.115 riassume la logica di sviluppo che ha nelle "Common Set API", ossia le API universali, il punto di forza per applicazioni scalabili. Il tutto rigorosamente funzionante su Windows 10.

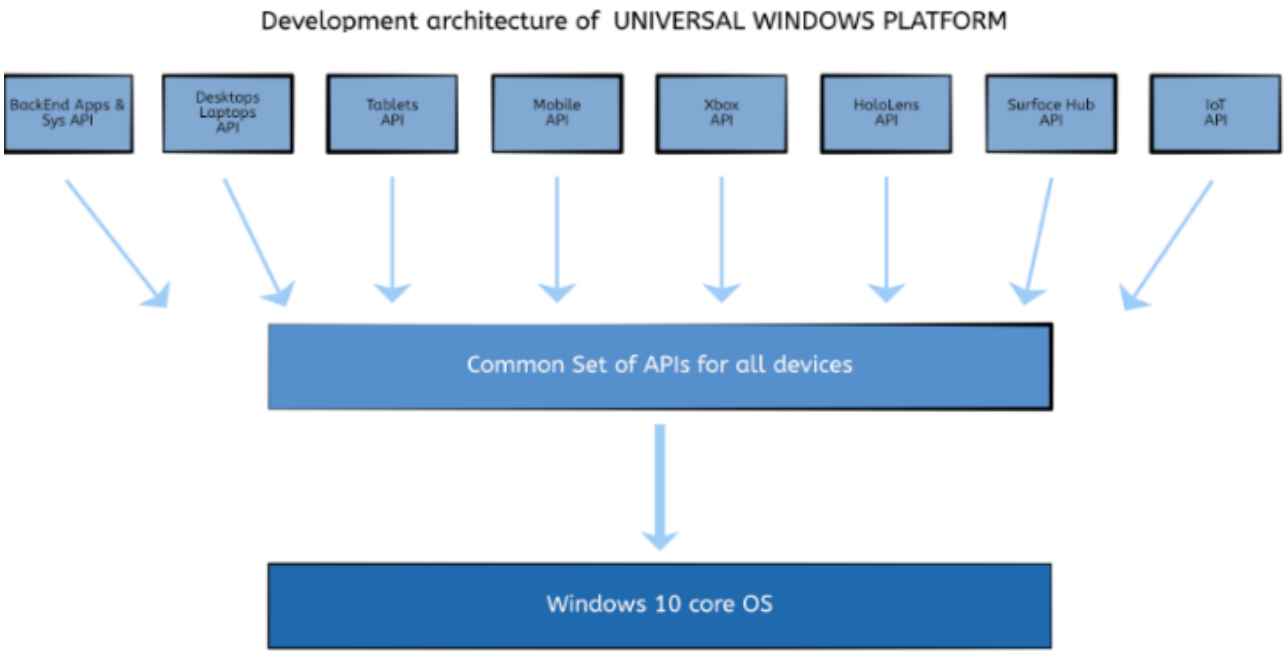


Fig.115

Un'unica applicazione UWP, un unico sistema operativo Windows 10, un unico Store per la distribuzione delle APP, un unico SDK che racchiude le API universali e quelle per le singole famiglie di device, un unico Azure come Cloud e, sorprendentemente, un unica interfaccia grafica la quale dovrà avere la caratteristica fondamentale di adattarsi sulla base della dimensione dello schermo e un'unica gestione dei dispositivi di input, come penna, microfono, ecc..

La realizzazione di interfacce grafiche nelle UWP avviene con l'ausilio di XAML (eXtensibile Application Markup Language) che è un linguaggio stile XML che permette di separare la logica applicativa da quella descrittiva sfruttando la grafica vettoriale.

La differenza tra la gestione grafica di una UWP e "Windows Form" classica è marcata, infatti le "Windows Form" utilizzano il sistema grafico GDI/GDI+ basato su User32 che gestisce a basso livello i pixel, con l'ovvia conseguenza che qualsiasi modifica a livello grafico viene gestita dall'applicazione stessa con conseguente sovraccarico e scarse

prestazioni. In tale contesto la scheda video non necessità di molta memoria, visto che è l'applicazione stessa che gestisce tutto. Questa modalità operativa prende il nome di "immediate". E' interessante notare, come da Fig.116, che l'applicazione "Windows Form" per modificare l'aspetto grafico, necessità di ottenere una richiesta formale da parte del sistema operativo, tramite il messaggio "WM_PAINT", quindi nel caso il programmatore gestisca l'evento "OnPaint" di un pulsante o altro oggetto grafico, sarà l'applicazione stessa a richiedere al sistema operativo, tramite un messaggio di invalidazione, di ricevere il messaggio "WM_PAINT". Una volta ricevuto il messaggio l'applicazione interagisce direttamente con il sistema grafico GDI/GDI+.

Con la potenza delle schede video moderne, la modalità "immediate" è obsoleta oltre che scarsamente performante, quindi è stata sostituita da un sistema grafico più potente come WPF, il quale è basato su grafica vettoriale e si appoggia alle DirectX. La modalità operativa di WPF è chiamata "retained" ed ha il grande vantaggio che non è più l'applicazione che gestisce il rendering, ma il sistema grafico.

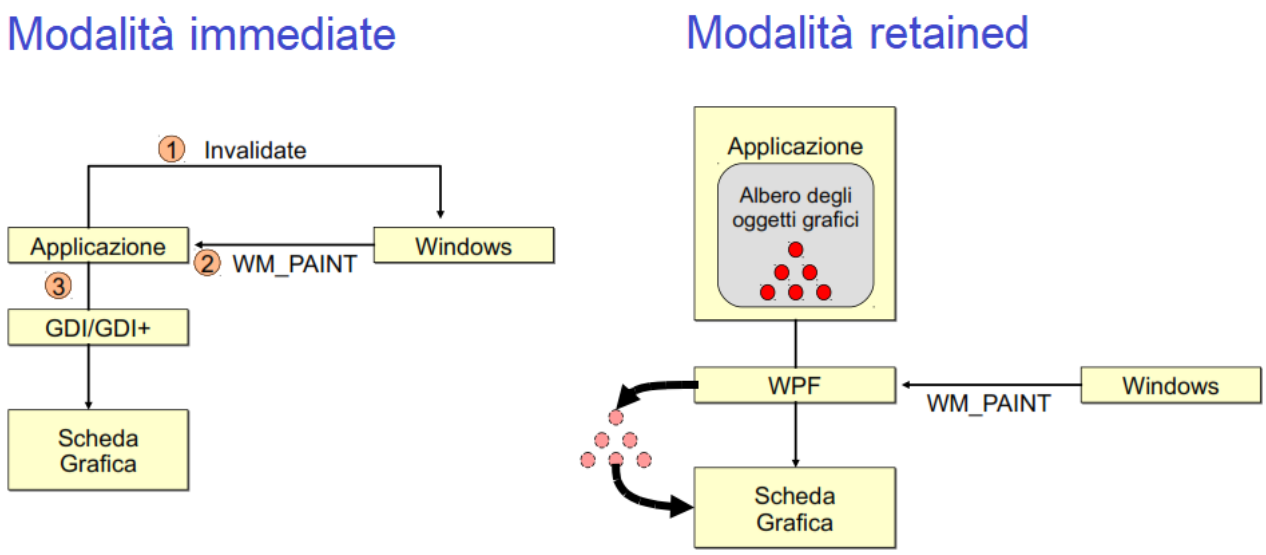


Fig.116

La gestione della grafica con WPF prevede l'esistenza di due Thread, il primo chiamato "UIThread" ha il compito di effettuare l'aggiornamento dei componenti grafici e di acquisire l'input, il secondo chiamato "Rendering Thread", eseguito in background, ha il compito di inviare l'albero degli oggetti grafici alla scheda video. Questa gestione degli oggetti grafici, tramite il linguaggio XAML, garantisce delle performance molto elevate, ma un quantitativo

di memoria a bordo della scheda video nettamente maggiore. Il vantaggio visivo nell'impiegare un sistema grafico basato sulla grafica vettoriale è chiaramente intuibile dalla Fig.117 che mostra come un immagine stirata subisca un effetto di "sgranatura", mentre tramite la grafica vettoriale comporti una scalatura che porta ad avere un immagine pulita. La grafica vettoriale permette l'indipendenza dalle caratteristiche hardware del monitor, ossia il sistema grafico impiega il DIP (Device Indipendent Pixel) che permette di lavorare con display di densità di pixel diversi.

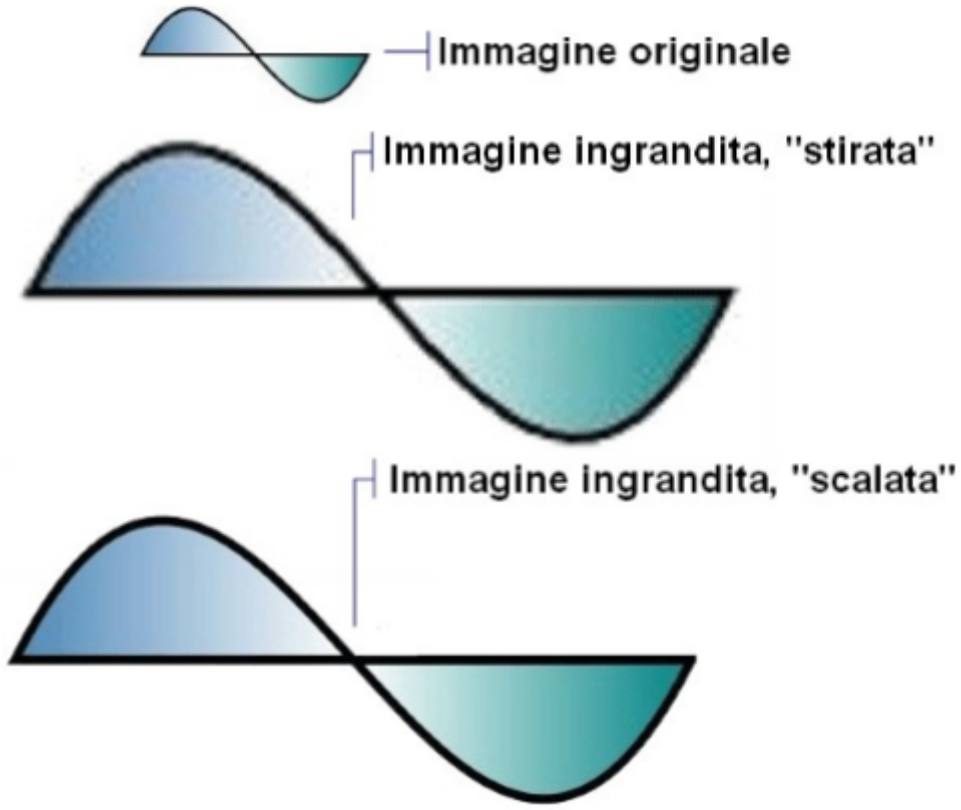


Fig.117

E' molto importante conoscere il modo di interazione tra lo "UIThread" e il "Rendering Thread" o comunque tra "UIThread" e thread secondari creati ad hoc perché nel momento in cui si andrà a modificare la UWP col fine di testare la pressione di un bottone, con conseguente stampa a video di un messaggio, entrerà in gioco il legame tra thread che possiede l'oggetto e thread secondario che ne vuole modificare la proprietà. Ogni applicazione grafica possiede un Dispatcher, ossia una coda dei messaggi a priorità (sulla falsa riga del Dispatcher Win32) nella quale il "UIThread" inserisce le richieste al verificarsi

di un evento, come ad esempio la pressione di un bottone (sia tramite touch che tramite penna), oppure legge i messaggi di notifica arrivati da altri thread. Il "Rendering Thread" preleva dal Dispatcher le richieste di inizio attività e inserisce l'esito delle attività svolte. La Fig.118 mostra il Dispatcher di WPF ed il possibile impiego dei metodi "Invoke()" e "BeginInvoke()". Il metodo "Invoke()" è bloccante, quindi attende il messaggio di risposta da parte del "Rendering Thread", mentre il metodo "BeginInvoke()" è asincrono, quindi permette al codice di continuare l'esecuzione di altre istruzioni, ma obbligherà l'impiego della "EndInvoke()" qualora sia necessario venire a conoscenza dell'esito delle attività svolte, in caso contrario non sarà necessaria tale chiamata.

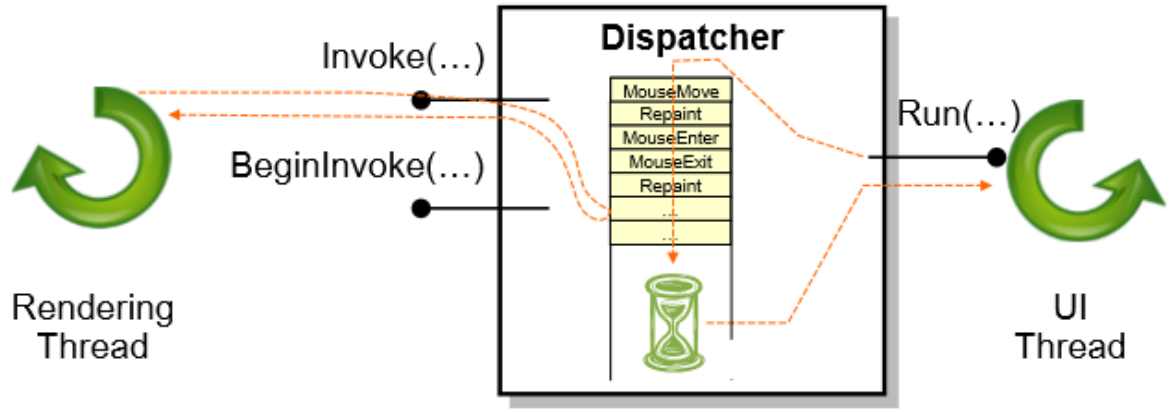


Fig.118

UWP utilizza quindi come API universali parte del motore grafico di WPF, quindi molte cose viste in questo paragrafo hanno riscontro anche in questa architettura, in primis l'uso della grafica vettoriale, l'uso del linguaggio XAML e l'impiego del Dispatcher. Ovviamente WPF è un framework, introdotto dal .NET 3.0, sviluppato appositamente per la famiglia dei dispositivi desktop, quindi potrà offrire delle caratteristiche a più basso livello per tali device. L'applicazione UWP che verrà sviluppata per il testing del Pi con diodo led e pulsanti è molto banale, quindi non richiede a livello di gestione grafica l'interazione con device di input, come mouse, penna o joystick. Per tale ragione viene limitata all'impiego dei comuni elementi grafici senza entrare nel dettaglio di tutte le problematiche di adattamento dei componenti, discussione più indicata per uno sviluppo intensive per famiglie di device come tablet e desktop, mentre per i dispositivi IoT di tipo headless, ossia senza schermo, può sembrare quasi inutile. In ogni caso conviene offrire un minimo di grafica anche per gli IoT così da garantire un primo step di debug visivo.

3.2.4 Creazione applicazione UWP con Visual Studio 2015

La presentazione del paragrafo precedente è servita per capire il significato che sta alla base della creazione di applicazioni universali, anche se una trattazione completa di questa infrastruttura richiederebbe libri di testo dedicati. Per creare una UWP aprire Visual Studio 2015 e dal menù "File" selezionare la voce "New" e poi "Project..." come da Fig.119.

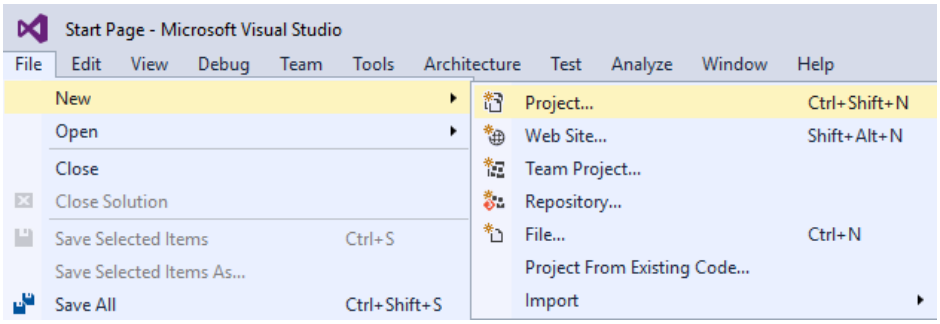


Fig.119

Selezionare le voci "Visual C#", "Windows", "Universal" e creare un progetto "Blank App (Universal Windows)". Specificare il nome delle Location e il nome del progetto (Name).

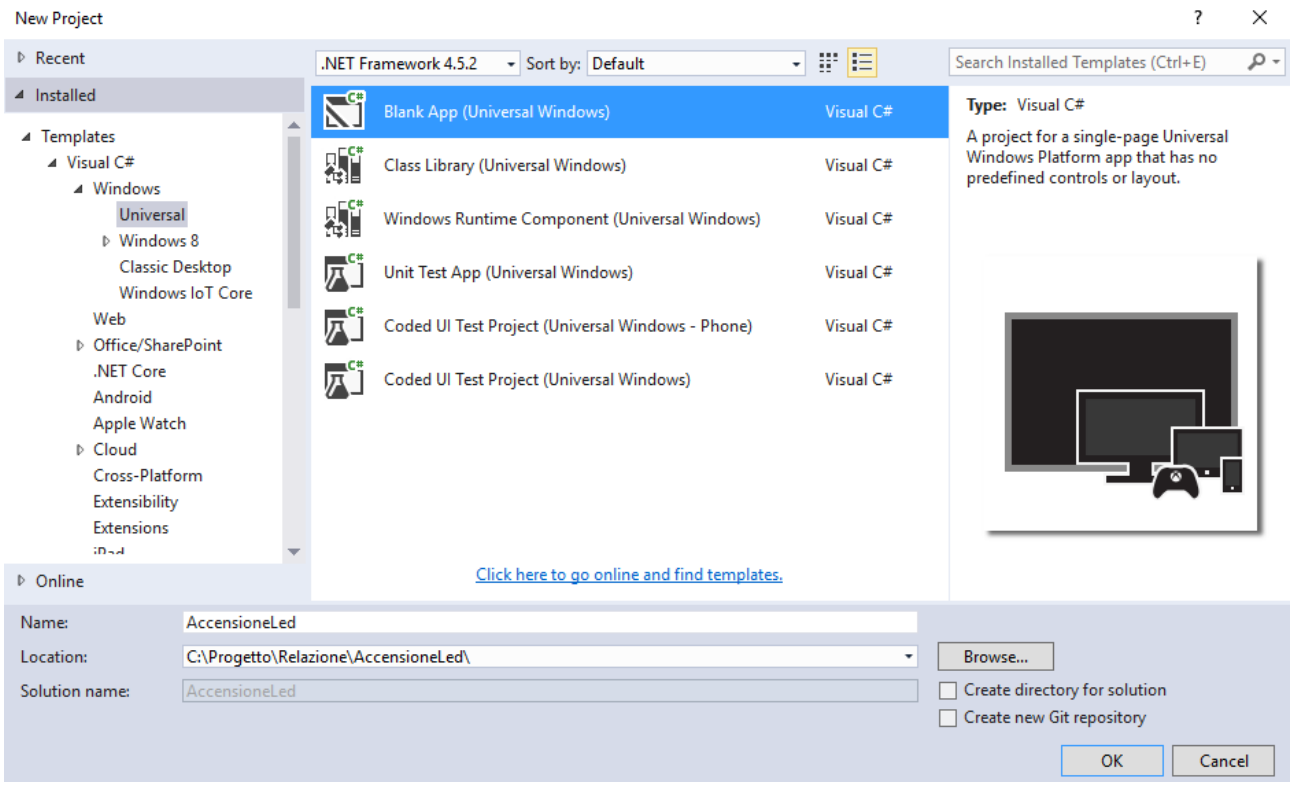


Fig.120

Confermare la creazione dell'applicazione premendo il pulsante "OK". Lo step successivo è capire l'utilità della "Solution Explorer" di Fig.122, la quale potrebbe essere stata chiusa in precedenza, di conseguenza è possibile riattivarla tramite l'opzione "Solution Explorer" del menù "View", come da Fig.121. Può essere comodo nella "Solution Explorer" utilizzare l'icona a forma di puntina, così che la finestra si chiuda o si apra automaticamente ad ogni click sull'apposita voce. Si veda la Fig.123.

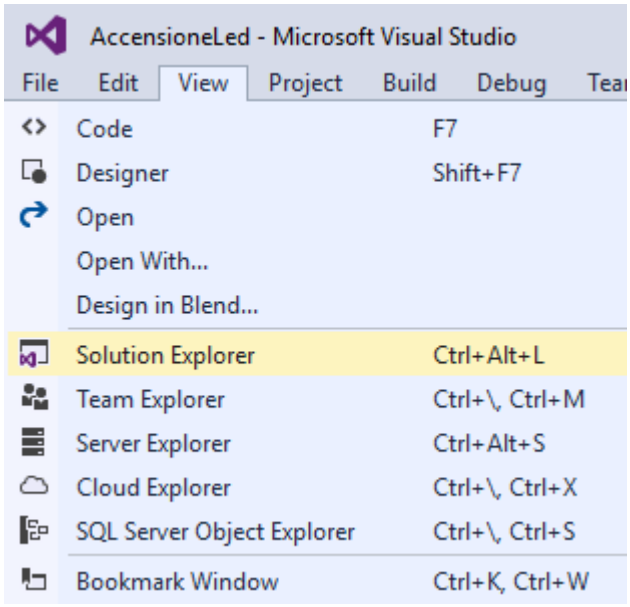


Fig.121

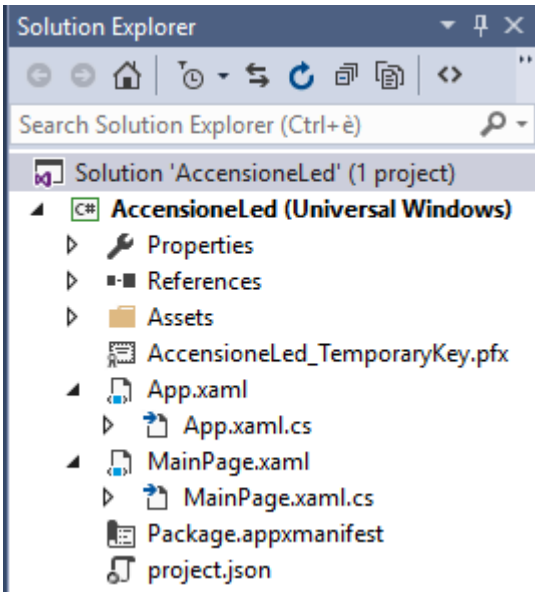


Fig.122

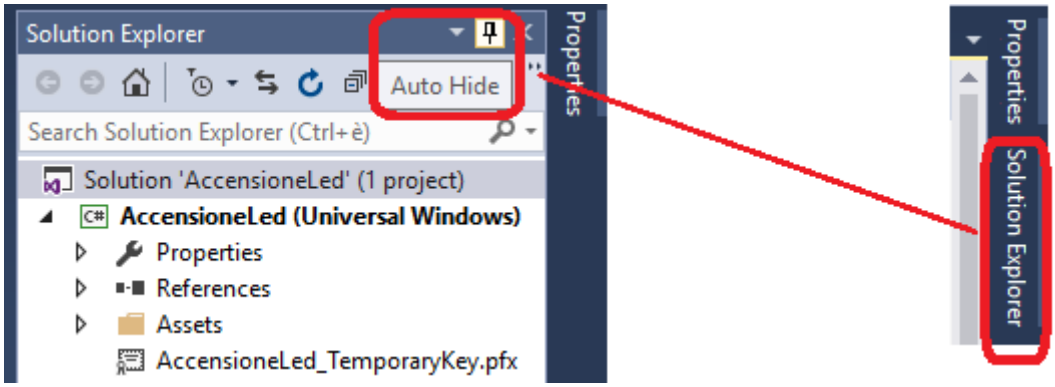


Fig.123

Il file progetto su cui lavorare a livello grafico è "MainPage.xaml" mentre a livello di codice C# è "MainPage.xaml.cs". L'impostazione grafica di base prevede la possibilità di scrivere direttamente codice XAML oppure utilizzare il Drag&Drop dei componenti. Per passare da una modalità all'altra basta utilizzare le voci Design e XAML di Fig.124 che prevede in

primis la seleziona del file "MainPage.xaml" con un doppio click nello Solution Explorer di Fig.122. La Fig.125 riassume questi passi per lavorare sul file di progetto.

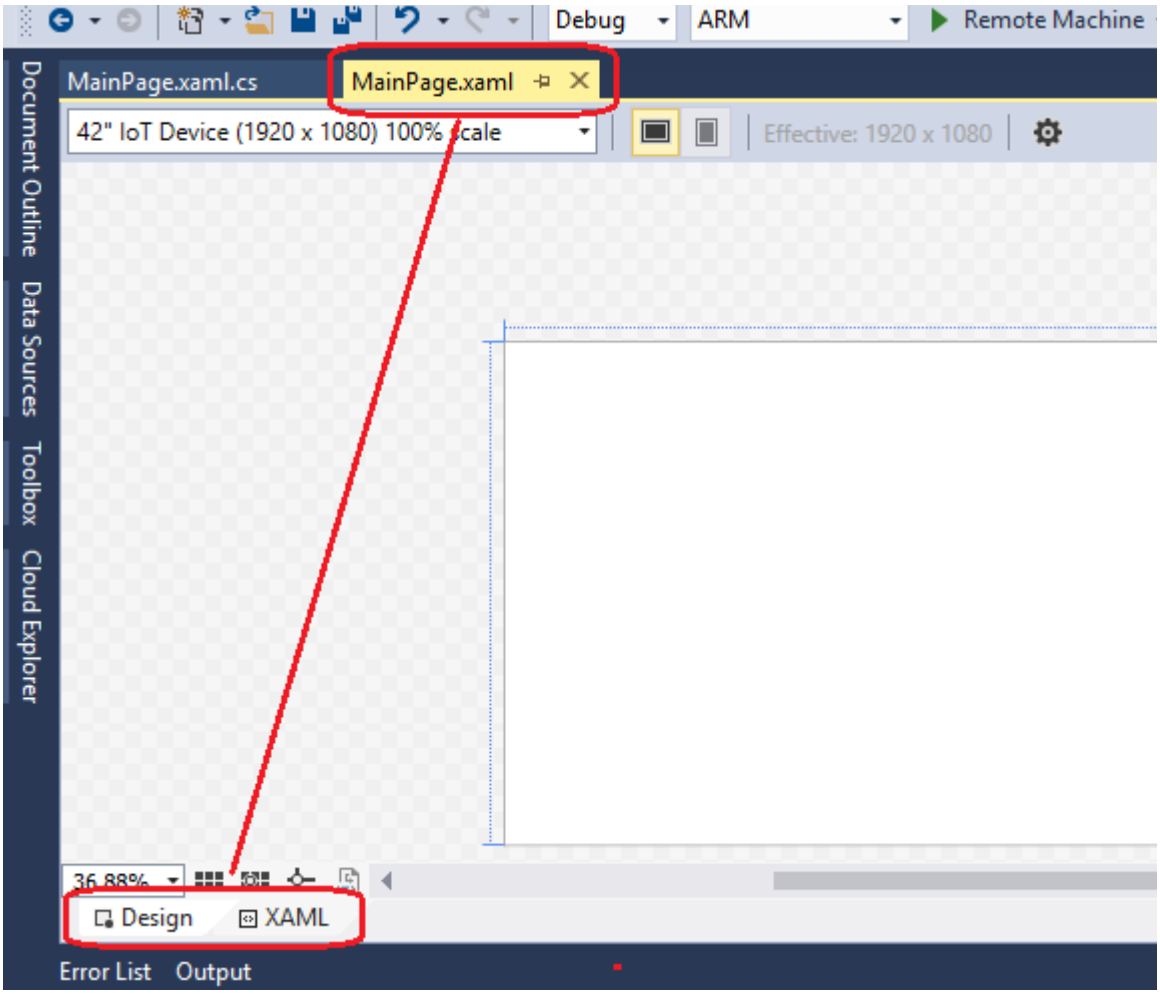


Fig.124

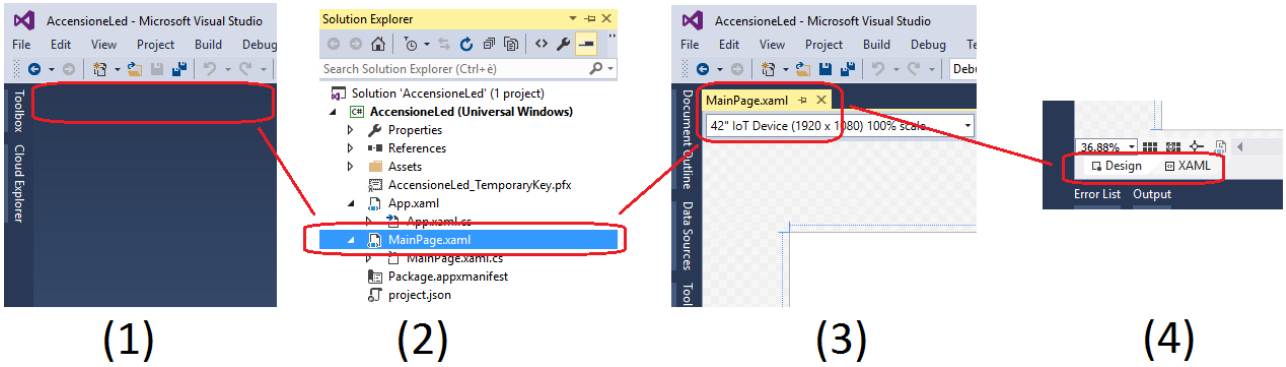


Fig.125

La trattazione su come elaborare una UWP adattabile per ogni famiglia di device va oltre lo scopo di questa discussione, in ogni caso è interessante evidenziare quali sono gli strumenti adattativi che si possono utilizzare. I due componenti classici sono lo "StackPanel" che ordina i propri elementi figli in sequenza verticale oppure orizzontale e la "Grid" che opera in modo equivalente ad una griglia CSS inserendo gli elementi figli all'interno delle celle. Il limite di questi due pannelli di layout è la capacità di adattamento, motivo per cui la Microsoft ha introdotto un nuovo componente per la definizione dello stile di layout chiamato "RelativePanel" che non utilizza la logica di annidamento degli oggetti grafici, ma un loro corretto riposizionamento. Questo approccio permette di ottenere applicazioni che si adattano molto meglio alle diverse risoluzioni dello schermo. Per inserire i componenti grafici è necessario selezionare la "Toolbox" dal menù "View" come in Fig.126. Il risultato di questa operazione è l'inserimento nella parte sinistra del Visual Studio della apposita sezione con tutto l'elenco dei componenti che si possono impiegare. Si veda la Fig.127.

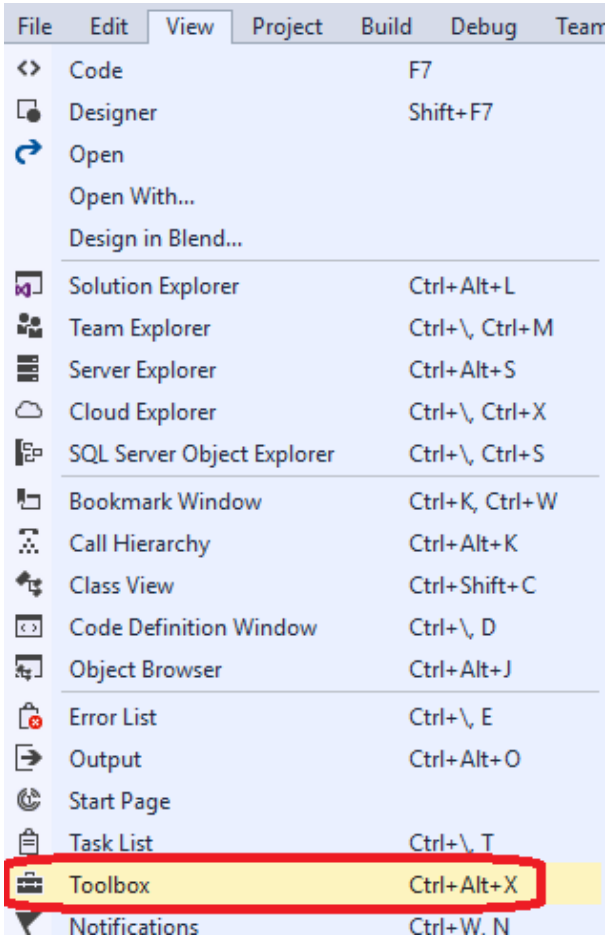


Fig.126

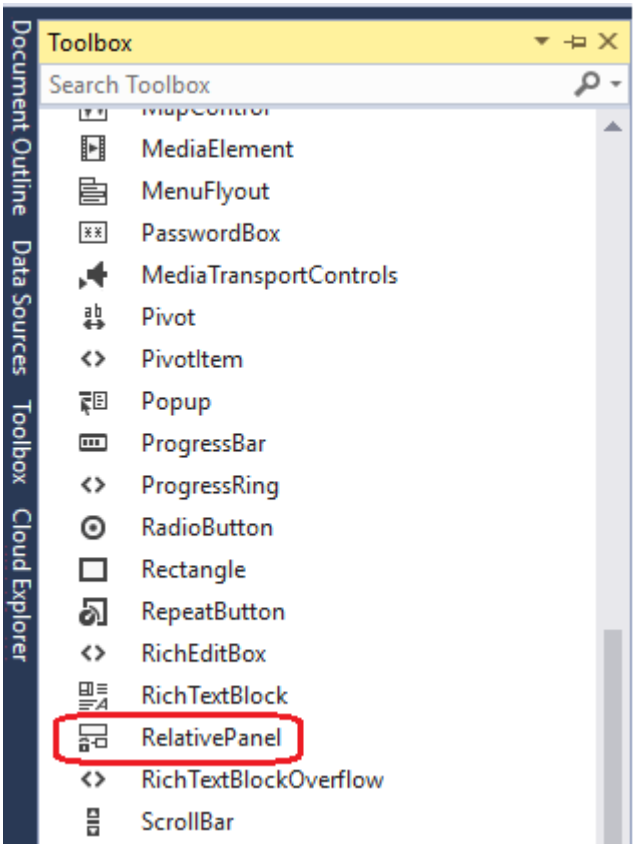


Fig.127

Impostare la dimensione della UWP direttamente in fase di Design selezionando la voce "42 Iot Device" visto che il Raspberry Pi 3 è collegato ad un monitor 48", ma in qualsiasi momento è possibile vedere l'adattamento ad altre risoluzioni variando questa voce. La Fig.128 imposta la risoluzione di 42".

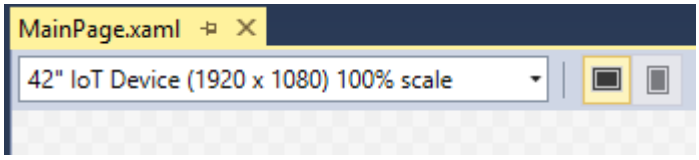


Fig.128

Lo steps successivo è abbozzare un design grafico nel workspace vuoto, come si vede in Fig.129, trascinando come primo componente il "RelativePanel" e dentro di esso tre "TextBox" nelle quali è possibile inserire del testo. La realizzazione grafica è tutta qui, infatti non è lo scopo di questo progetto creare APP adattative per una variegata famiglia

di device, considerando poi che i dispositivi IoT sono quasi sempre senza schermo. Il risultato finale è visibile in Fig.130.

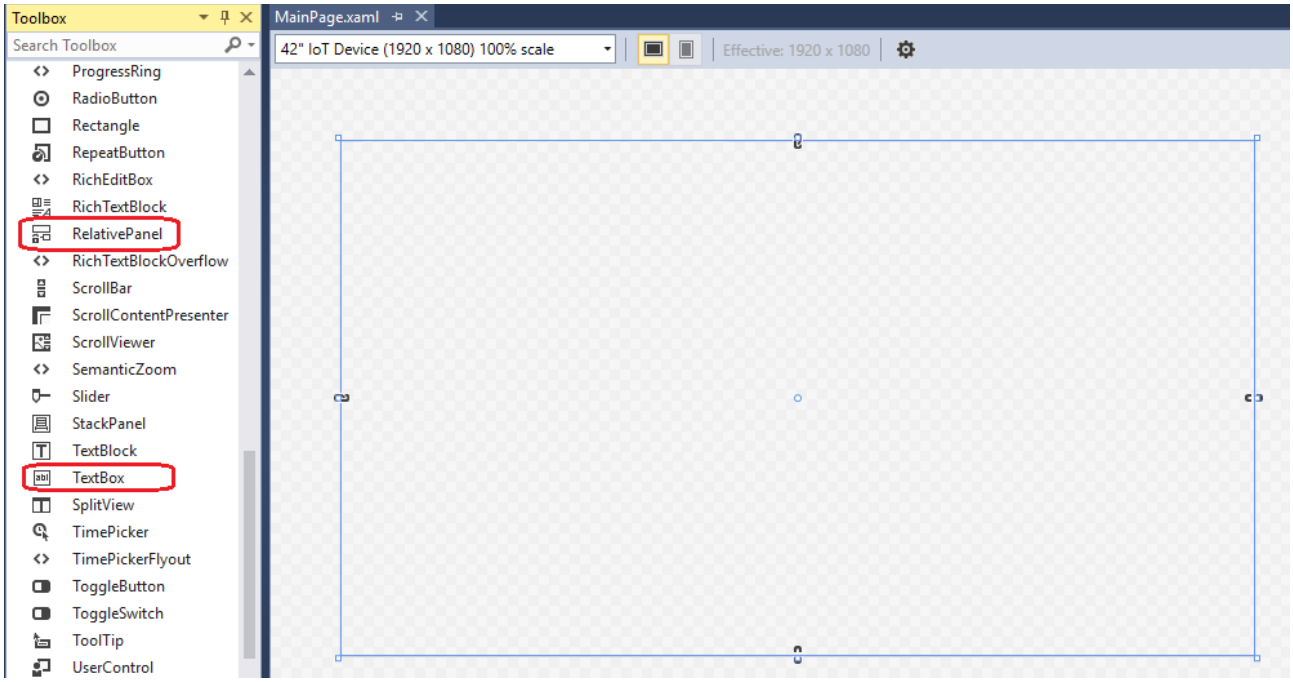


Fig.129

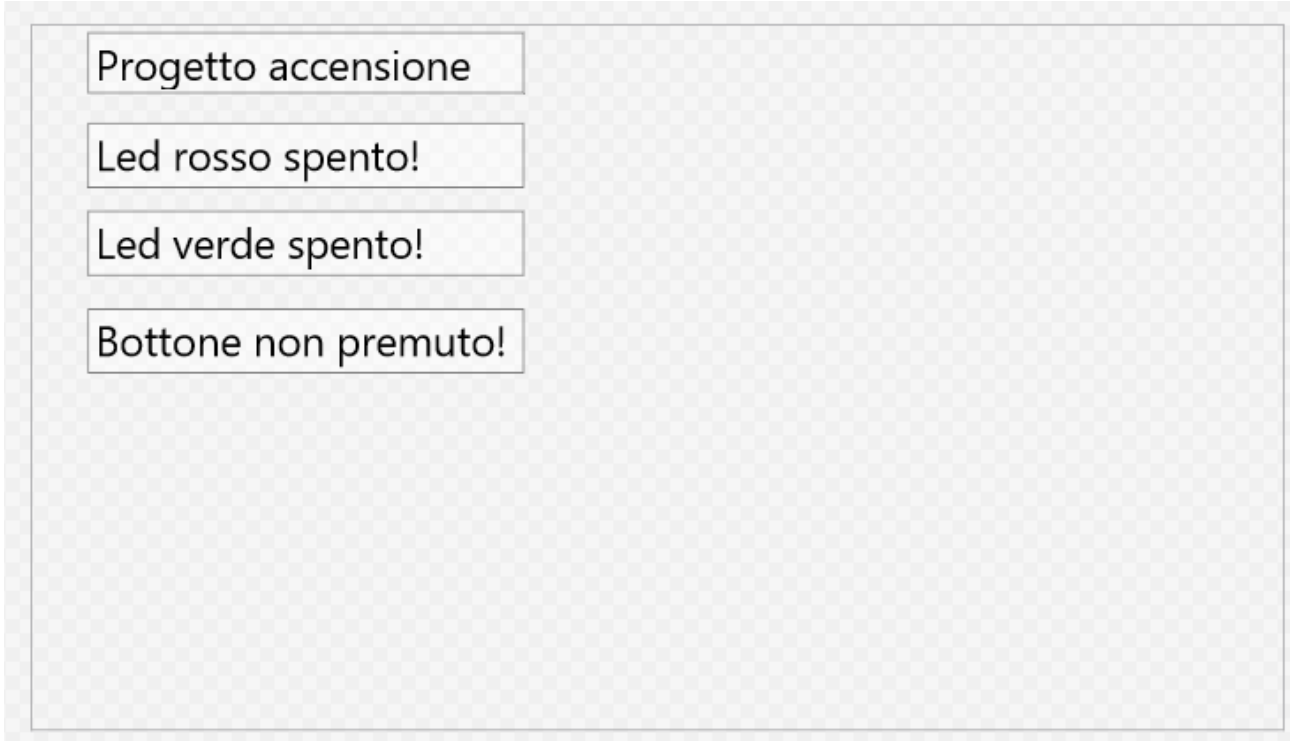


Fig.130

La personalizzazione del contenuto del testo può essere fatta comodamente via grafica, sfruttando la sezione "Proprieties" che si abilita premendo il tasto funzione F4 o la relativa voce dal menù "View". Una volta attivata la "Proprieties", selezionando una delle TextBox o la RelativePanel è possibile impostare le varie proprietà, come si vede in Fig.131.

Fig.131

Viene scelto il nome "txtBenvenuto" per la prima TextBox tramite l'attributo "Name", mentre il contenuto del testo è facilmente impostabile tramite l'attributo "Text" che però non è visibile nella Fig.131. Un modo più semplice per interagire con i componenti, è lavorare direttamente in XAML, anche se questo a prima vista può sembrare lungo è complicato. La Fig.132 mostra la visualizzazione finale del contenuto XAML di questa semplice UWP, mentre sotto è riportato il codice completo sul quale è opportuno fare una brevissima discussione.

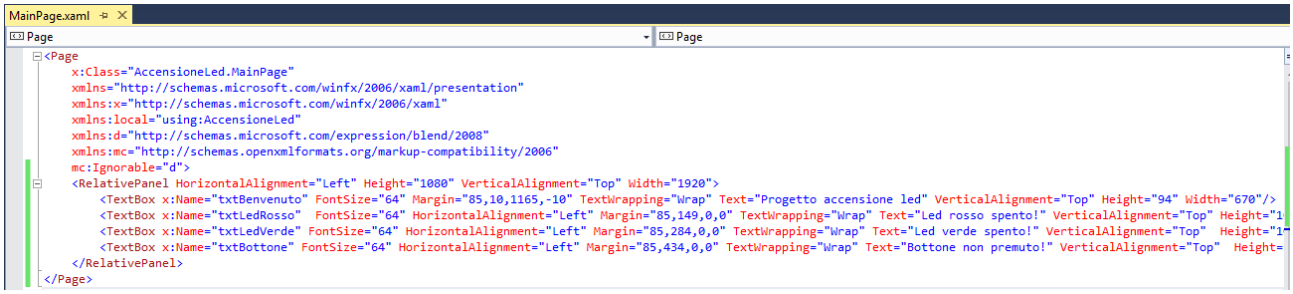


Fig.132

La struttura del file XAML ricalca quella di XML, con un tag <Page attr></Page> che contiene tutti gli elementi grafici ed un insieme di attributi. Il componente <RelativePage></RelativePage> contiene a sua volta gli elementi grafici ai fini

dell'adattamento. Le modifiche che sono state apportate vertono essenzialmente sulle tre TextBox, alle quali è stato cambiato il nome, definito un nuovo attributo "FontSize=64", impostata la distanza di 85 pixel dal margine sinistro, il relativo testo nell'attributo "Text" e una lunghezza di 670 pixel per un altezza del componente pari a 100 pixel. In rosso sono state indicate tutte le modifiche, che come si vede sono minime e abbastanza semplici.

```
<Page
   x:Class="AccensioneLed.MainPage"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:local="using:AccensioneLed"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
                  mc:Ignorable="d">
   <RelativePanel HorizontalAlignment="Left" Height="1080" VerticalAlignment="Top"
                                                  Width="1920">
      <TextBox x:Name="txtBenvenuto" FontSize="64" Margin="85,10,1165,-10"
                                TextWrapping="Wrap" Text="Progetto accensione led"
                                VerticalAlignment="Top" Height="100" Width="670"/>
      <TextBox x:Name="txtLedRosso"  FontSize="64" HorizontalAlignment="Left"
                                Margin="85,149,0,0" TextWrapping="Wrap"
                                Text="Led rosso spento!" VerticalAlignment="Top"
                                Height="100"  Width="670"/>
      <TextBox x:Name="txtLedVerde" FontSize="64" HorizontalAlignment="Left"
                                Margin="85,284,0,0" TextWrapping="Wrap"
                                Text="Led verde spento!" VerticalAlignment="Top"
                                Height="100" Width="670"/>
      <TextBox x:Name="txtBottone" FontSize="64" HorizontalAlignment="Left"
                                Margin="85,434,0,0" TextWrapping="Wrap"
                                Text="Bottone non premuto!" VerticalAlignment="Top"
                                Height="100" Width="670"/>
   </RelativePanel>
</Page>
```

Terminata l'impostazione grafica, comprensiva anche della TextBox del bottone, è necessario procedere alla scrittura del codice C#, selezionando il file "MainPage.xaml.cs" direttamente con un doppio click nella "Solution Explorer" come indicato in Fig.133.

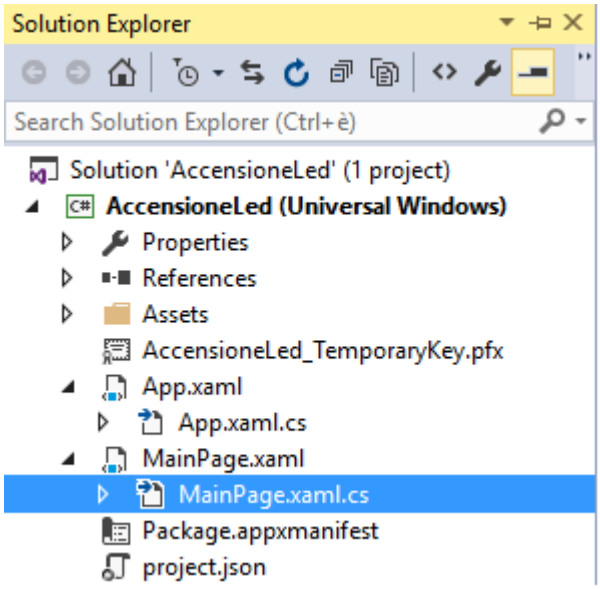


Fig.133

La scrittura del codice può seguire molti paradigmi di programmazione, dal classico MVC (Model-View-Controller) alla IoC (Inversion of Control - Dependency Injection). In questo piccolo progetto si vuole mettere in risalto il modus operandi per interagire con i pin delle GPIO, quindi vengono lasciati da parte gli approcci operativi a queste tecniche di programmazione. Prima di scrivere una sola riga di codice è necessario inserire una libreria aggiuntiva appartenente alla "Universal Windows Extension" chiamata "Windows IoT Extension for the UWP". Per inserire librerie aggiuntive selezionare la voce "References" del progetto nella "Solution Explorer" e col tasto destro del mouse scegliere "Add Reference" come si vede in Fig.134. Una volta aperto il "Reference Manager" selezionare la voce "Extensions" dal menù "Universal Windows" e selezionare l'opzione "Windows IoT Extension for the UWP" costruzione "10.0.10240.0" come da Fig.135. Confermare la scelta tramite il pulsante "OK". Il risultato a video è l'inserimento della libreria nei riferimenti di progetto, come indicato in Fig.136. Molto probabilmente si è notato la presenza della stessa libreria in due costruzioni diverse, diviene quindi essenziale capire la differenza tra le varie release sulla base delle indicazioni di Microsoft. La Tab.8 riporta la differenze principali tra i vari build.

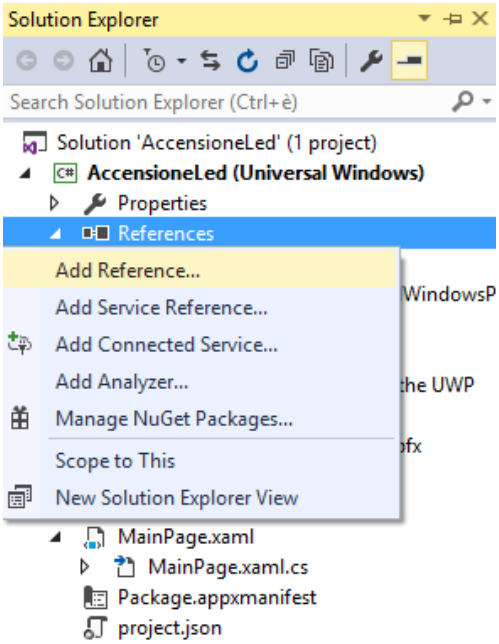


Fig.134

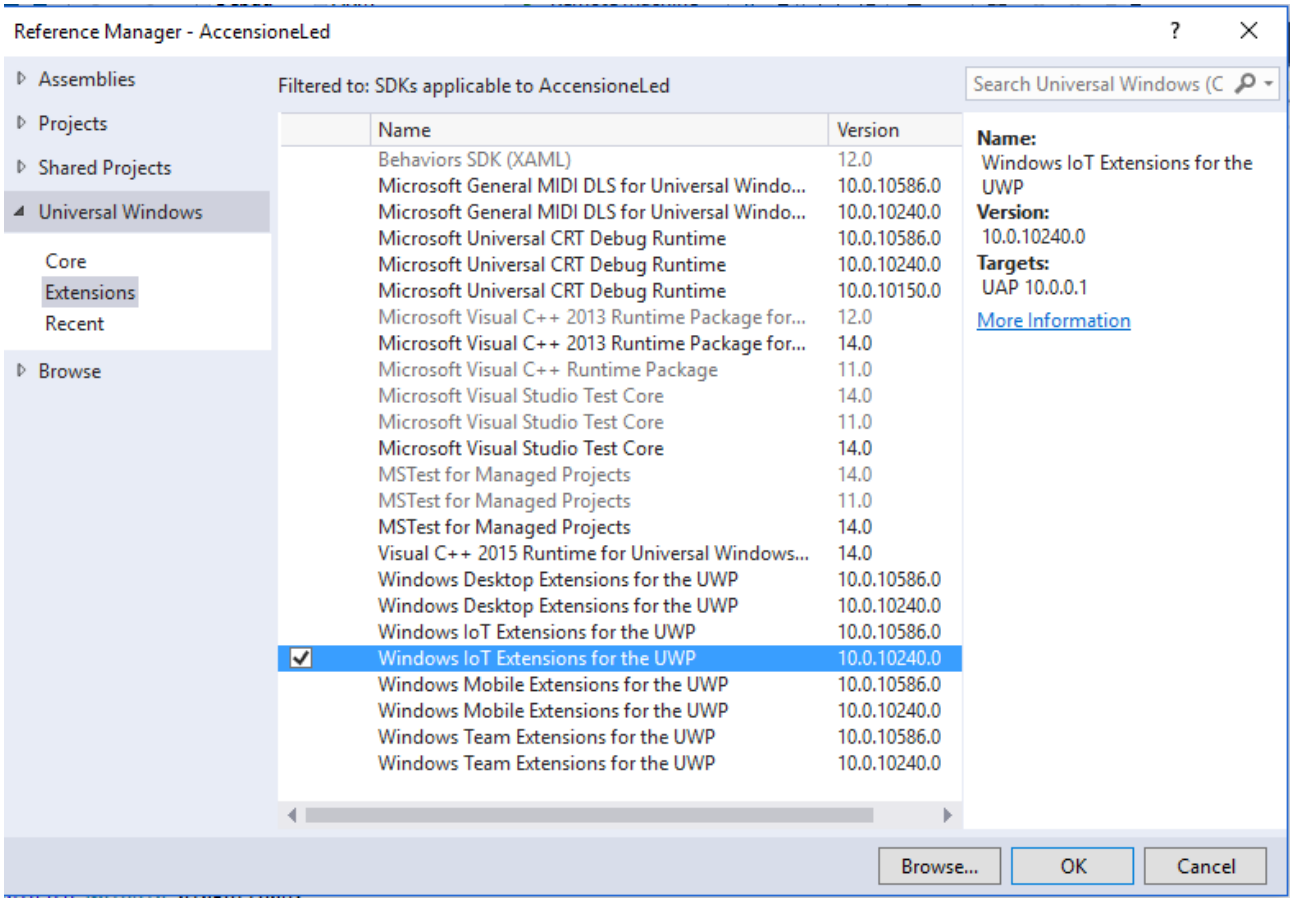


Fig.135

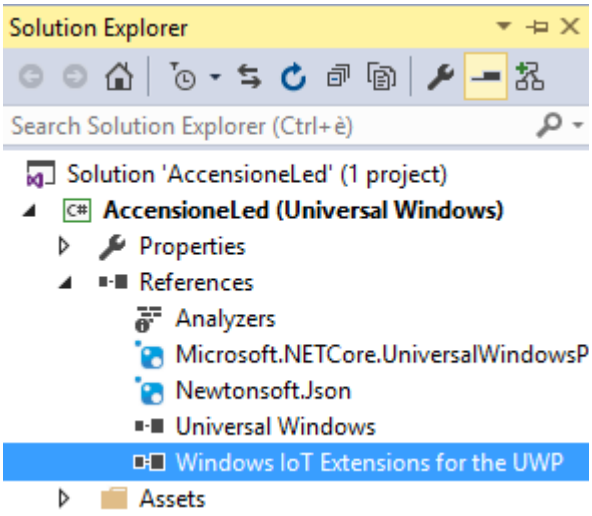


Fig.136

Versione	Descrizione
14393 (Anniversary Edition)	versione rilasciata nel luglio 2016 che include tre nuove feature come "Windows Ink", "Cortana API" e "Windows Hello" che permette a Microsoft Edge il riconoscimento biometrico.
10586	versione rilasciata nel novembre 2015 che introduce ORTC (Object Real Time Communication) API per comunicazioni video in Microsoft Edge
10240	versione base di Windows 10 luglio 2015

Tab.8

Per semplificare le cose si è preferito utilizzare come "IoT Extension API" la versione base 10240, quindi conviene impostare esclusivamente tale release nelle proprietà di progetto. Per fare questo selezionare il nome di progetto nella "Solution Explorer" come in Fig.137.

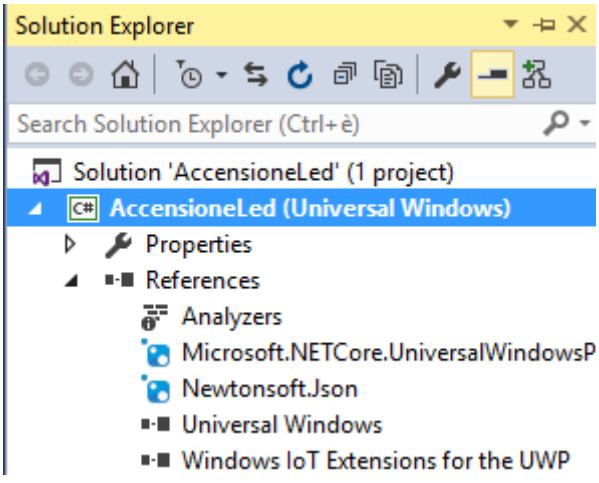


Fig.137

Successivamente premere il tasto destro del mouse e selezionare la voce "Proprieties" come in Fig.138.

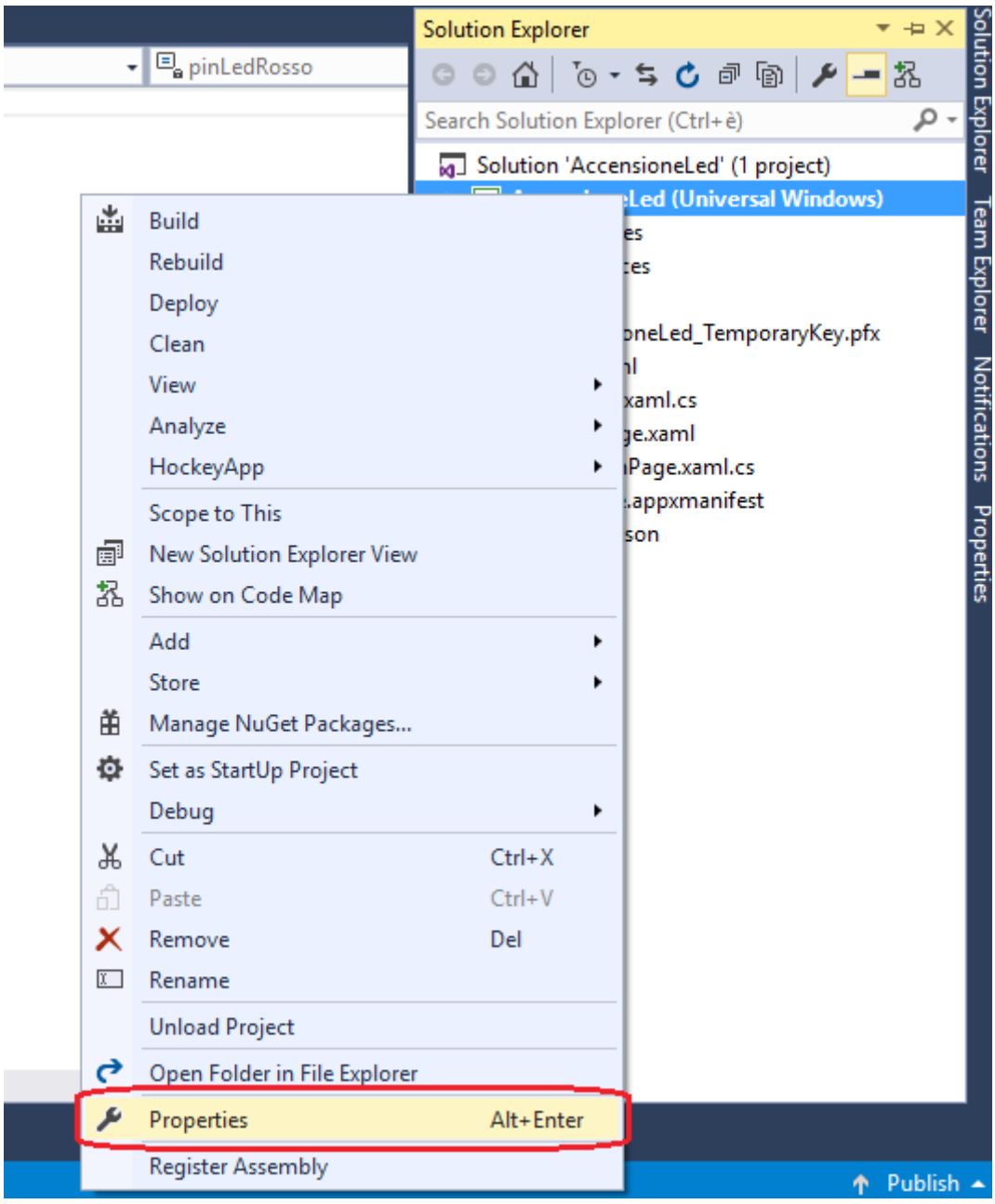


Fig.138

Nella sezione "Application" verificare che la "Target version:" e la "Min version:" sia impostata con la costruzione "10240" come in Fig.139.

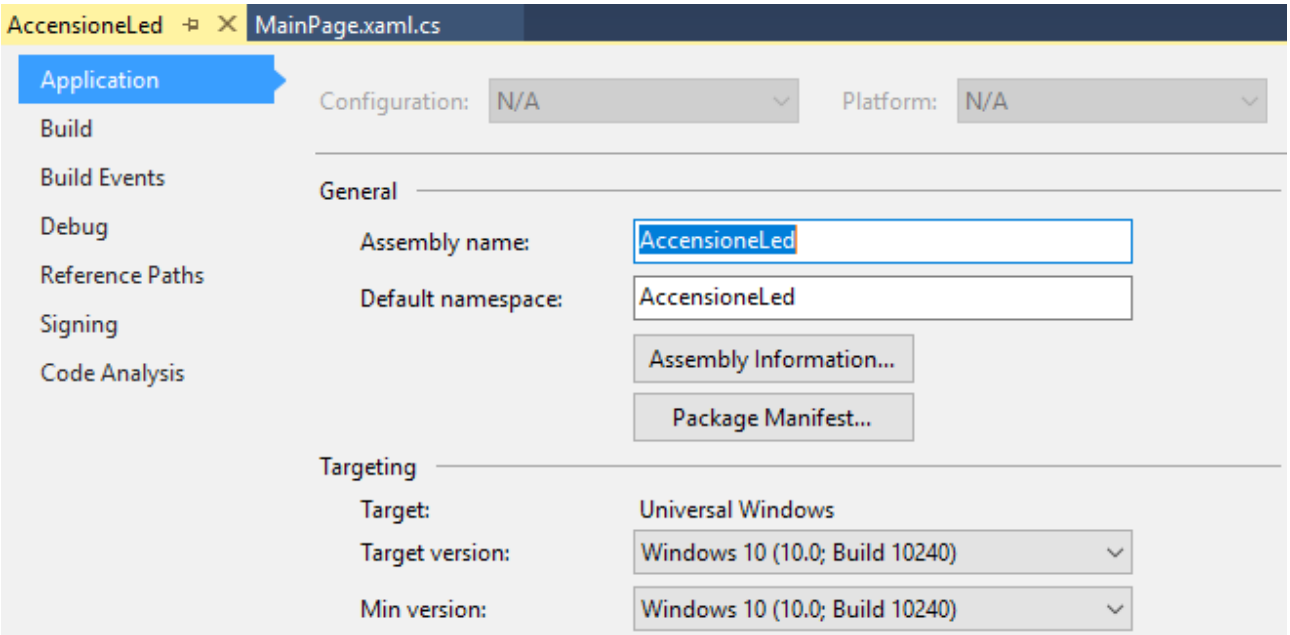


Fig.139

Per lavorare con la GPIO è necessario utilizzare la classe GpioController() presente nel reference "Windows IoT Extension for the UWP". Per evitare di qualificare in modo esteso la classe è consigliabile dichiarare la direttiva "using" sotto quelle già presenti nel progetto. La riga di codice da inserire è la seguente:

```
using Windows.Devices.Gpio;
```

All'interno della classe "MainPage" definire delle costanti intere che rappresentano la GPIO utilizzata secondo lo schematics di Fig.107 senza eliminare nessuna riga di codice già presente nel progetto. Il valore da assegnare alle costanti è quello relativo al numero della GPIO e non alla numerazione del pin, come accade ad esempio nel linguaggio Python. I pin usati sono la GPIO23 e GPIO25 quindi si utilizzano gli scalari 23 e 25.

```
public sealed partial class MainPage : Page
{
    private const int pinLedRosso = 23;
    private const int pinLedVerde = 25;

     ...

}
```

All'interno del metodo "MainPage()", che inizializza tutti gli elementi grafici tramite il metodo "this.InitializeComponent()", inserire un nuovo metodo chiamato ad esempio "AccendiLed()".

```
public MainPage()
{
    this.InitializeComponent();
    AccendiLed();
}
```

L'implementazione di questo metodo è molto semplice e per comodità di esposizione si sono indicati dei numeri in verde utili per commentare il significato della corrispondente riga di codice. Il metodo ha un ambito di visibilità di tipo privato e non restituisce nulla.

```
private void AccendiLed()
{
        var gpio = GpioController.GetDefault();              // 1)

        if(gpio==null)                                       // 2)
        {
            txtBenvenuto.Text = "Device senza GPIO!";

            return;
        }
        else
        {
            var pinRosso = gpio.OpenPin(pinLedRosso);        // 3)
            pinRosso.Write(GpioPinValue.High);               // 4)
            pinRosso.SetDriveMode(GpioPinDriveMode.Output);  // 5)
            txtLedRosso.Text = "Led rosso accesso!";         // 6)

            var pinVerde = gpio.OpenPin(pinLedVerde);
            pinVerde.Write(GpioPinValue.Low);                // 7)
            pinVerde.SetDriveMode(GpioPinDriveMode.Output);
            txtLedVerde.Text = "Led verde accesso!";
        }
}
```

La riga 1) alloca un dato locale "var" di tipo "auto", ossia si lascia il compito al CLR assegnare il tipo corretto, tipo che dipende dal risultato della chiamata "GpioController.GetDefault()" che restituisce un oggetto "GpioController()" se la scheda embedded possiede una GPIO, "null" in caso contrario. In sostanza nella "var" sarà presente un handle per l'interazione con il periferico GPIO.
La riga 2) effettua un semplice test tramite il controllo condizionale "if" e nel caso "var" sia uguale a "null", viene stampato nella "txtBenvenuto" un messaggio di avvertimento ed il programma ritorna.
Il blocco di istruzioni dalla riga 3) alla riga 6) serve per impostare la parte di circuito con il led rosso, circuito che permetterà l'accensione del diodo led per passaggio di corrente se e solo se alla GPIO23 verrà applicata una differenzia di potenziale di +3.3V, che in logica binaria equivale ad uno stato alto.
La riga 3) definisce un dato locale "pinRosso" che rappresenta la GPIO23 grazie al metodo "OpenPin()" al quale si passa la costante "pinLedRosso" precedentemente definita.
La riga 4) definisce sul questo pin, tramite il metodo "Write()", un livello logico alto che comporterà di conseguenza un livello di tensione pari a +3.3V. Per definire il livello logico alto si utilizza per comodità la costante "High" della classe "GpioPinValue" che vale 1.
La riga 5) serve per definire il ruolo del pin del Raspberry Pi, ossia è necessario specificare chi comanda il livello di tensione di +3.3V, ruolo che cade ovviamente sul Pi il quale fornisce tensione al circuito, quindi il ruolo è di "Output". Per specificare il ruolo si utilizza la costante "Output" della classe "GpioPinDriveMode" che vale 1. Tale costante va inserita nel metodo che imposta il ruolo ossia il "SetDriveMode()".
La riga 6) stampa all'interno della "txtLedRosso" un messaggio di accensione del led sfruttando l'attributo "Text".
Il blocco di istruzioni successive serve per impostare la parte di circuito con il led verde, maglia elettrica già connesso al potenziale più alto di +3.3V, ma priva del potenziale di massa di 0V, valore di tensione che dovrà essere fornito in uscita dal pin GPIO25, motivo per cui anche il ruolo di questo pin è di "Output".
L'unica differenza è nell'istruzione 7) che prevede di fornire alla maglia una tensione di 0V così che la corrente possa circolare ed accendere il led. Per fare questo si deve indicare la costante "Low" della classe "GpioPinValue" che vale 0.

Terminato il programma è necessario impostare la piattaforma di compilazione corretta, visto che per default l'applicazione è configurata come x86. Oltre a questo è necessario specificare la macchina remota a cui inviare il progetto, macchina che sarà ovviamente il Raspberry Pi 3 accesso e correttamente connesso alla rete. Nella fase di configurazione era stato scelto l'indirizzo IP "192.168.0.100", in ogni caso conviene avviare la dashboard per essere sicuri che il Pi sia correttamente rilevato in rete, come fatto in Fig.140.

Fig.140

Una volta che vi è la certezza del corretto funzionamento in rete del Pi, si deve impostare l'ambiente corretto di compilazione e il deployment su macchina remota. Il primo step è modificare l'architettura di base da x86 a ARM, tramite l'apposito menù di Fig.141.

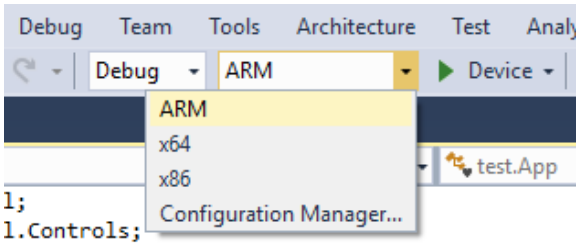


Fig.141

Nella sezione "Device" impostare la voce "Remote Machine" come in Fig.142.

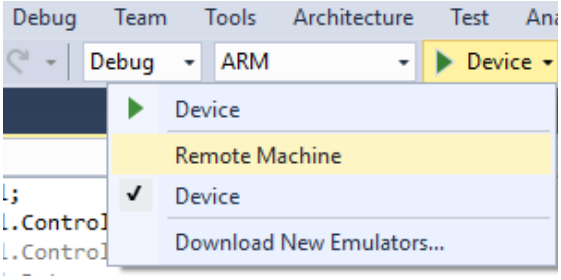


Fig.142

Il risultato è l'apertura della finestra di Fig.143.

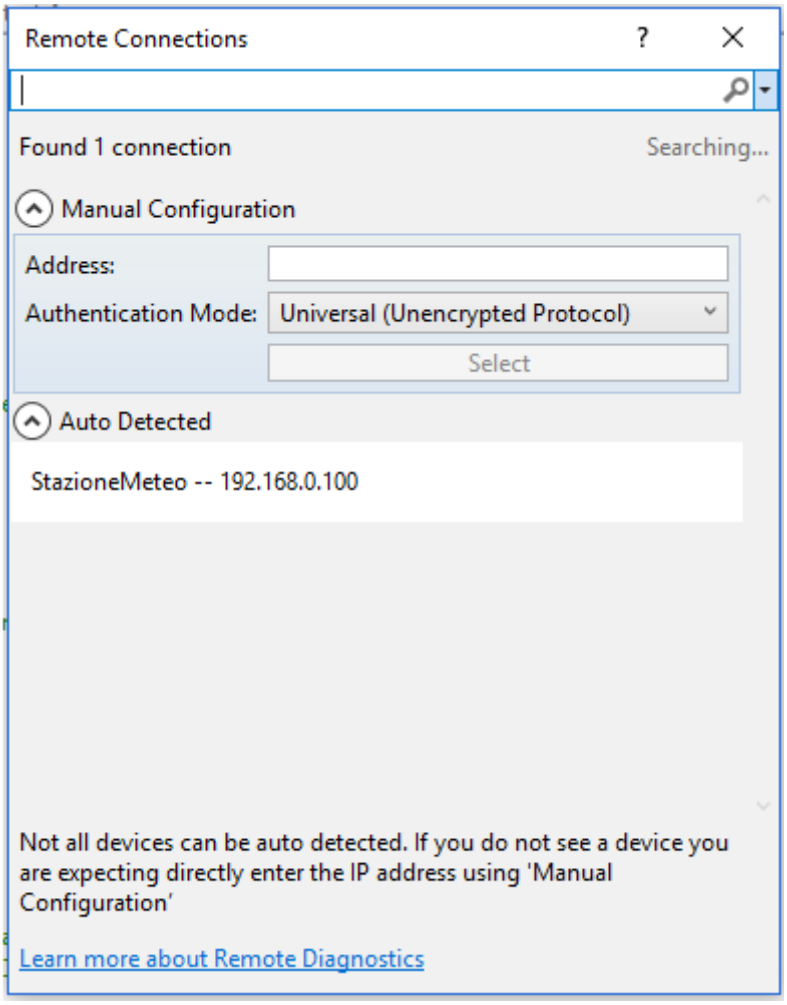


Fig.143

E' fondamentale impostare l'indirizzo IP del Raspberry nel campo "Address:", indirizzo che viene rilevato in automatico grazie all'auto detect, quindi selezionando il device "StazioneMeteo" si evita di inserire a mano le cifre che compongono l'indirizzo IPv4. La selezione del host "StazioneMeteo" riporta che nessun meccanismo di autenticazione è configurato per l'accesso al Pi, quindi qualsiasi malintenzionato potrebbe caricare la propria UWP direttamente sul device IoT. Realizzando in autonomia questo progetto, le problematiche di sicurezza non sono per il momento cardine principale, aspetto che però sarà trattato al termine del capitolo 4. La Fig.144 riporta il messaggio in questione, premere "Select" per impostare l'indirizzo e far si che la finestra si chiudi in automatico. Lasciare nel campo "Authentication Method" l'autenticazione proposta ossia "Universal (Unencrypted Protocol).

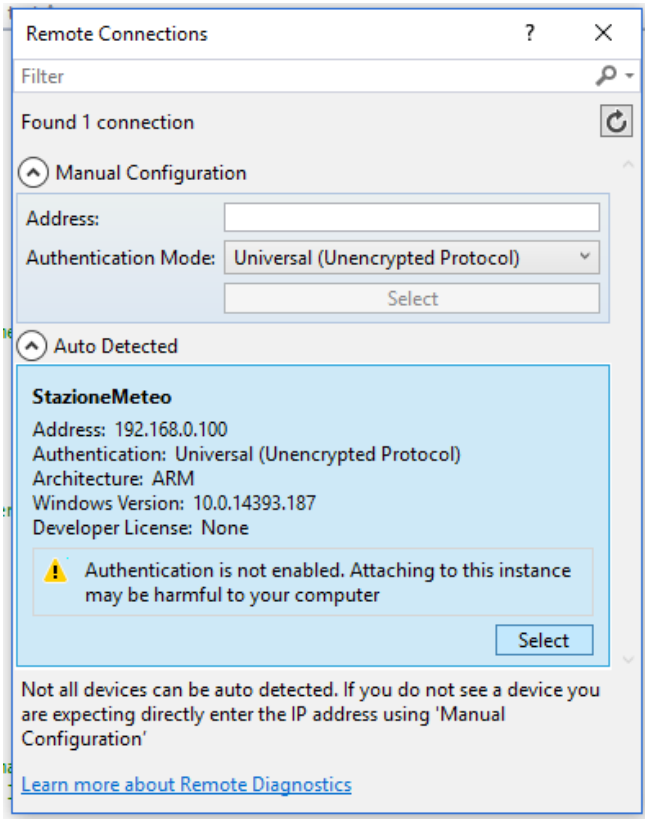


Fig.144

In qualsiasi momento è possibile vedere queste impostazioni, basta richiamare le "Proprieties" del progetto "AccensioneLed" come fatto in precedenza in Fig.137 e Fig.138 e nella sezione "Debug" verificare i parametri come si vede dalla Fig.145.

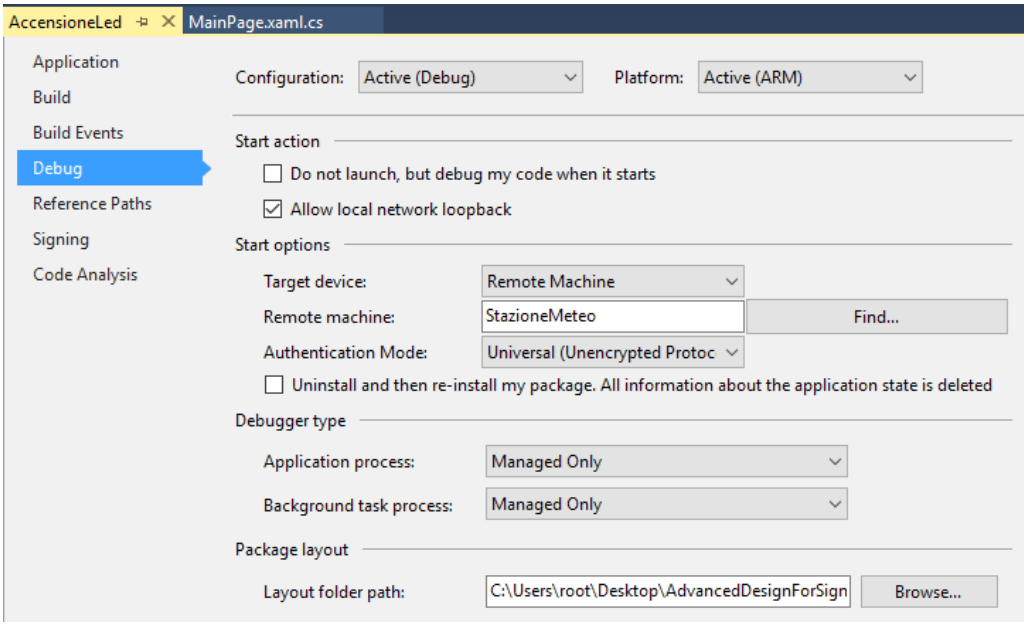


Fig.145

Terminata la fase di configurazione dell'ambiente Visual Studio, è possibile passare al deployment del progetto che esegue come primo step la compilazione del progetto e, nel caso non vi siano errori, procede al deployment verso il Raspberry Pi 3 con indirizzo IPv4 pari a "192.168.0.100". La fase di deploy si effettua tramite la voce "Deploy Solution" del menù "Build" come da Fig.146.

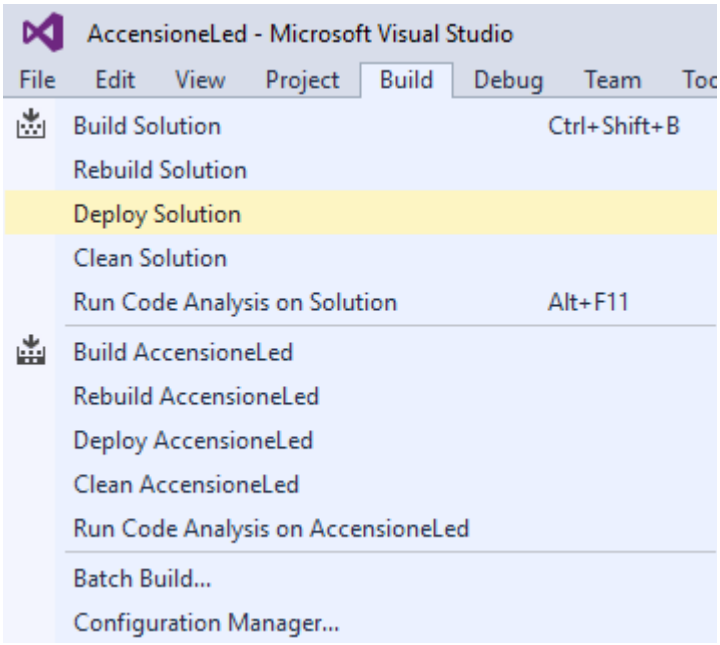


Fig.146

Il risultato del deployment è visibile in Fig.147, dove nella finestra di output sono riportate le due fasi, "Build" e "Deployment". Entrambe se eseguite correttamente devono riportare la dicitura "succeeded".
Risultato assai più importante è la presenza, dopo questa fase, della presenza della UWP "AccensioneLed" sul Raspberry Pi 3.

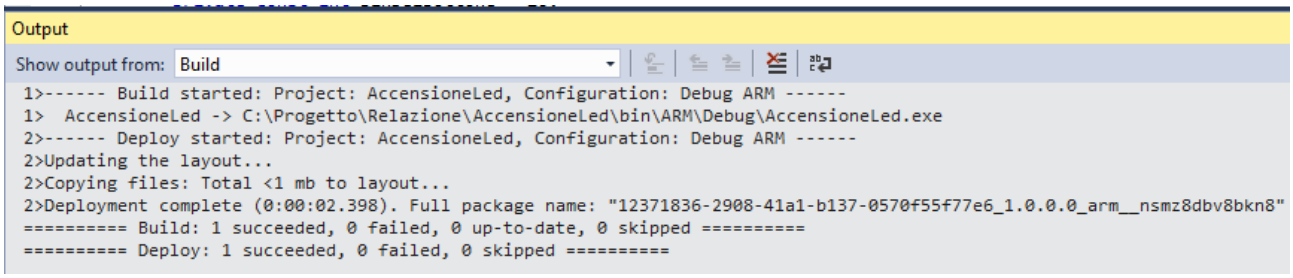


Fig.147

3.2.5 Avvio UWP su Raspberry Pi

La fase di progettazione hardware e software è terminata, ora è necessario effettuare il testing della UWP per riscontrare eventuali anomalie. Il paragrafo precedente è terminato con la copia dell'applicazione sul Pi, resta ora avviare la UWP è vedere il risultato, ossia i due led accesi con i rispettivi messaggi a video. Per avviare "AccendiLed" si deve accedere remotamente al Pi tramite browser, per fare questo potrete tranquillamente utilizzare la dashboard, in particolare l'opzione "Apri in Device Portal" di Fig.148 per poi effettuare l'autenticazione.

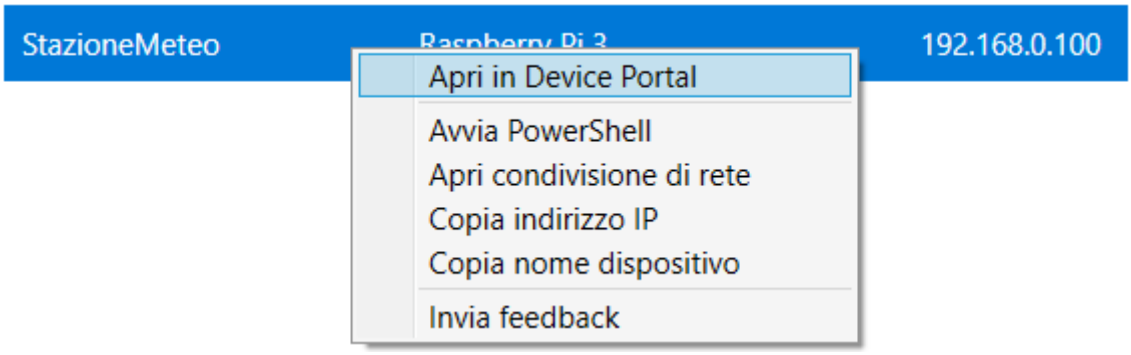


Fig.148

Dal browser selezionare la voce "Apps" così che si possano visualizzare tutte le applicazioni presenti sul device. Per avviare "AccensioneLed" basta premere sul pulsante a forma di start, come si vede in Fig.149. Per impostare l'avvio automatico dell'applicazione dopo il reboot del device premere il link "Set as Default App". Per rimuovere l'applicazione premere l'icona a forma di cestino.

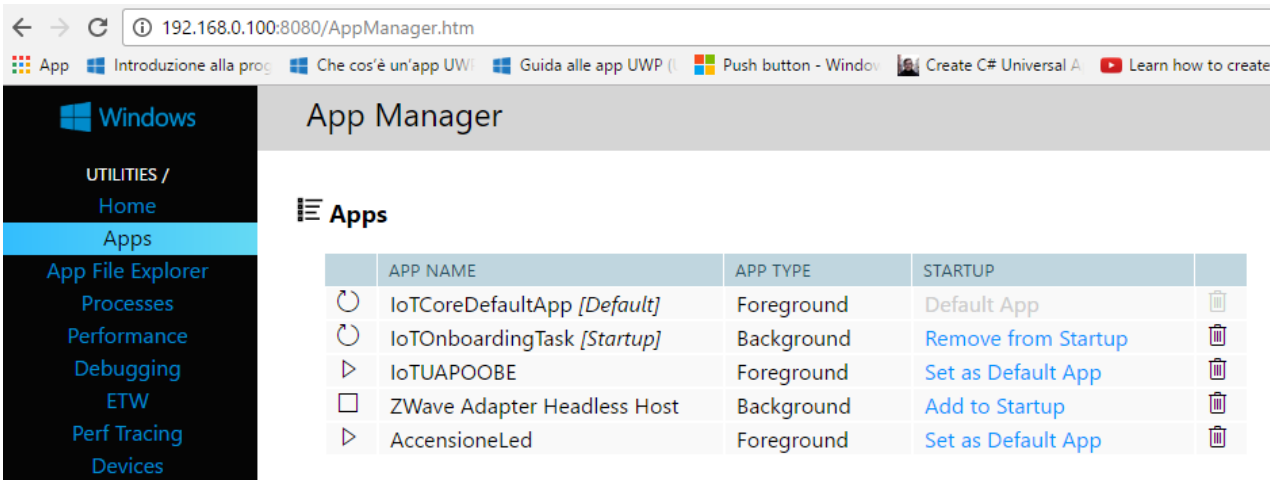


Fig.149

Il risultato dell'avvio dell'applicazione è visibile in Fig.150 e Fig.151.

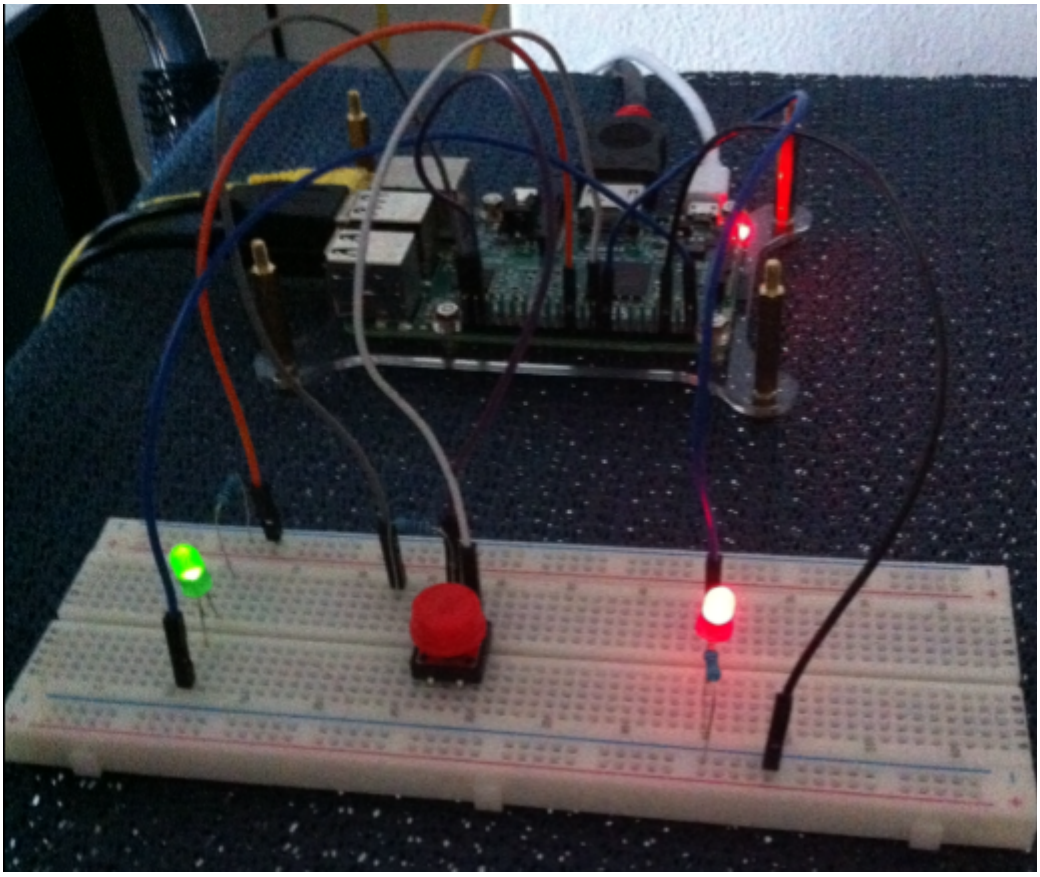


Fig.150

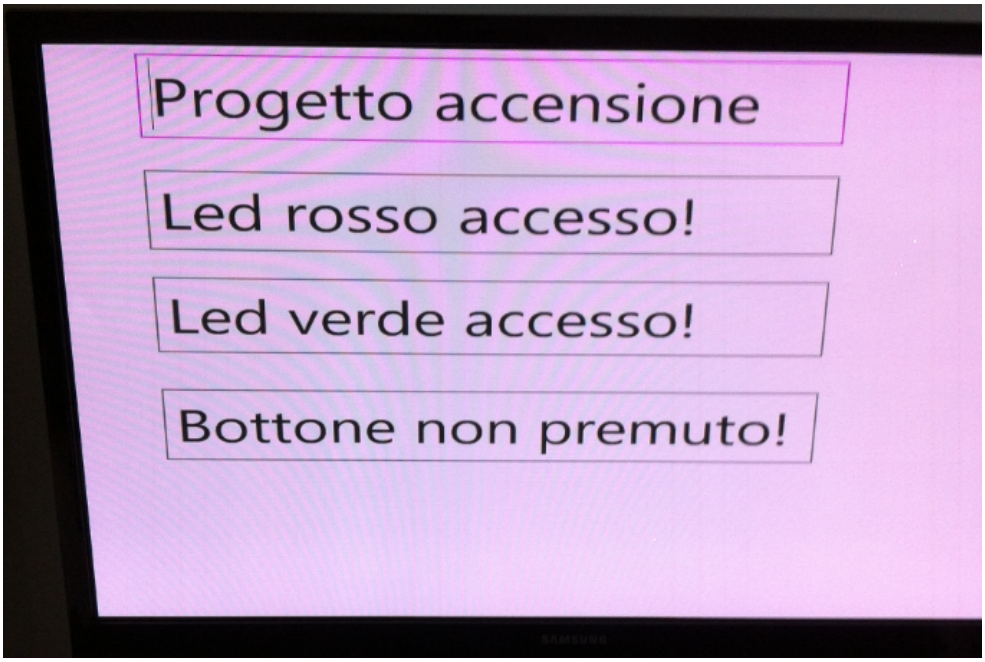


Fig.151

3.2.6 Integrazione pulsante

In questo paragrafo si è visto come interfacciare due semplici maglie elettriche con diodo led, decidendo poi il corretto livello di tensione in uscita sul pin. In verità, già in fase di design e XAML, si è impostato un'ulteriore Textbox per segnalare la pressione o meno di un pulsante, elemento circuitale che completa l'analisi di base di questo capitolo e che è già apparso in immagini precedenti. Il pulsante utilizzato è di tipo micro tattile di tipo J 12x12x5mm con capsula circolare rosso come si vede dalla Fig.152 e in Fig.150 montabile su bread board. Il pulsante è dimensionato per gestire al massimo 50mA a +12V.

Fig.152

Sulla falsa riga di quanto fatto per i led, è necessario progettare e capire la maglia elettrica, altrimenti diviene assai arduo capire in che modo programmare i pin GPIO del Raspberry Pi. In Fig.153 è riportato il progetto fritzing con l'inserimento di un solo pulsante, mentre in Fig.154 è stata riportata la sola parte relativa alla gestione del pulsante.

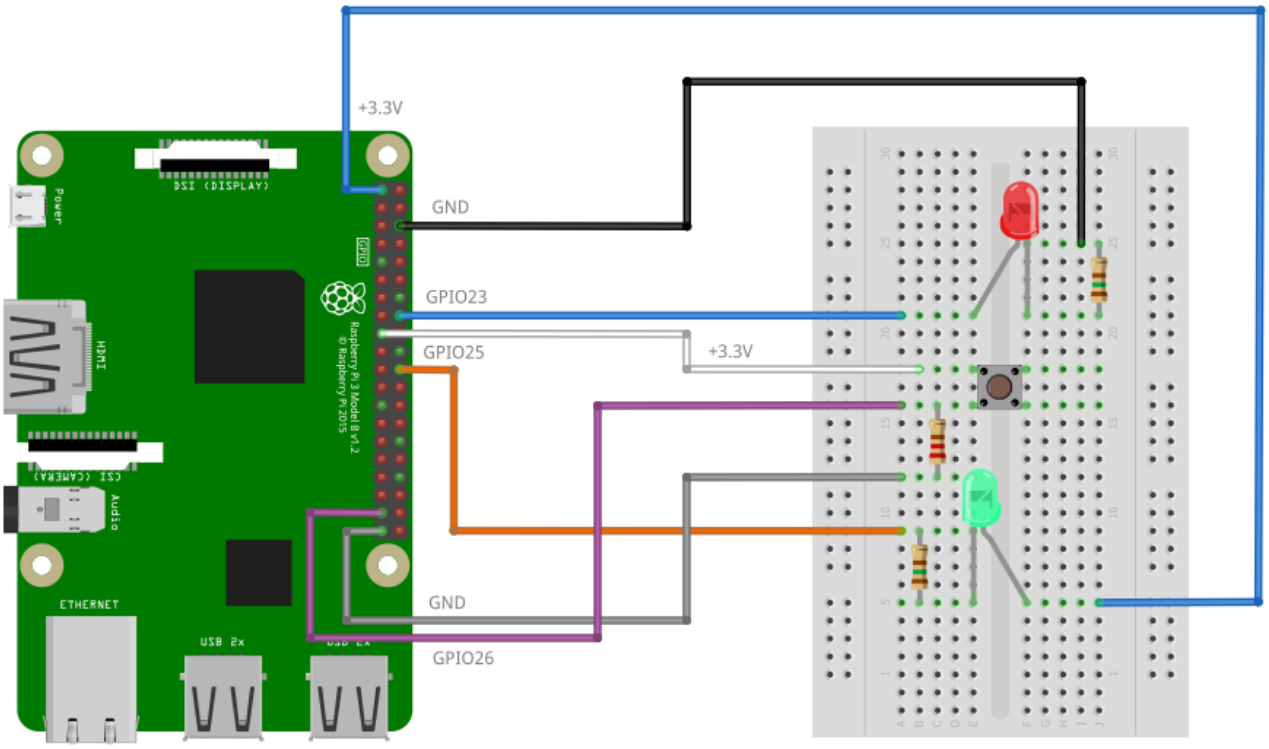


Fig.153

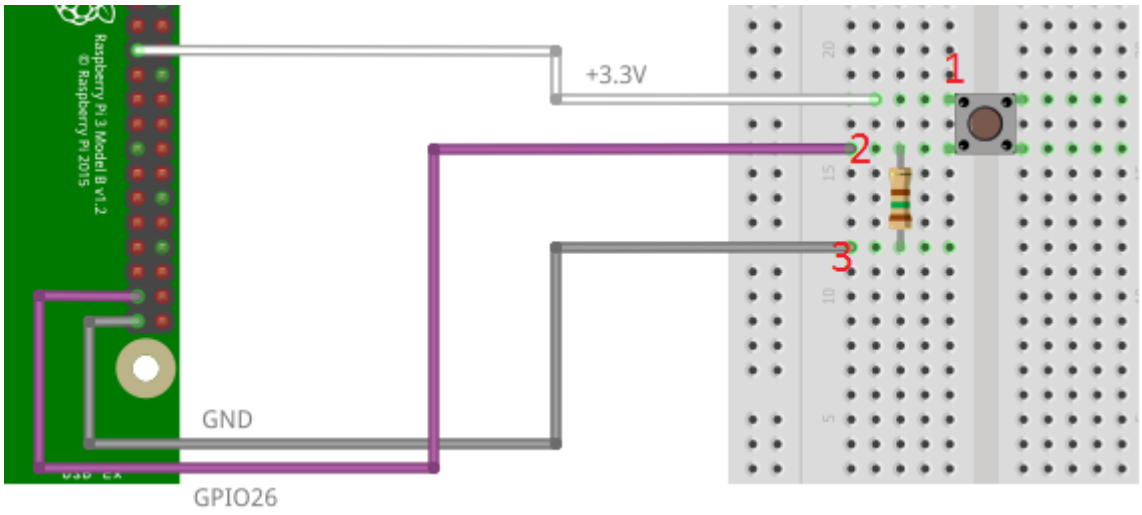


Fig.154

Il pin numero 17 porta la tensione di +3.3V, tramite il cavo di colore bianco, al contatto numero 1, ma non essendo il pulsante premuto, non si ha cortocircuito tra i contatti 1 e 2 , quindi non scorre corrente attraverso il resistore e la tensione al contatto 2 è nulla, ossia la GPIO26 sente un livello di tensione in entrata pari a 0V pari allo 0 logico. Nel momento in cui si preme il pulsante, si cortocircuitano i contatti 1 e 2, scorre corrente verso il potenziale di massa al punto 3 e quindi si forma una differenza di potenziale ai capi della

resistenza, che comporta la lettura di una tensione in entrata sulla GPIO26 pari a +3.3V, ossia 1 logico. Rispetto alle due maglie dei diodi led, il GPIO26 avrà un ruolo di entrata ai fini della lettura dello 0 o 1 logico e quindi della rispettiva tensione. La Fig.155 mostra lo schematics complessivo delle tre maglie.

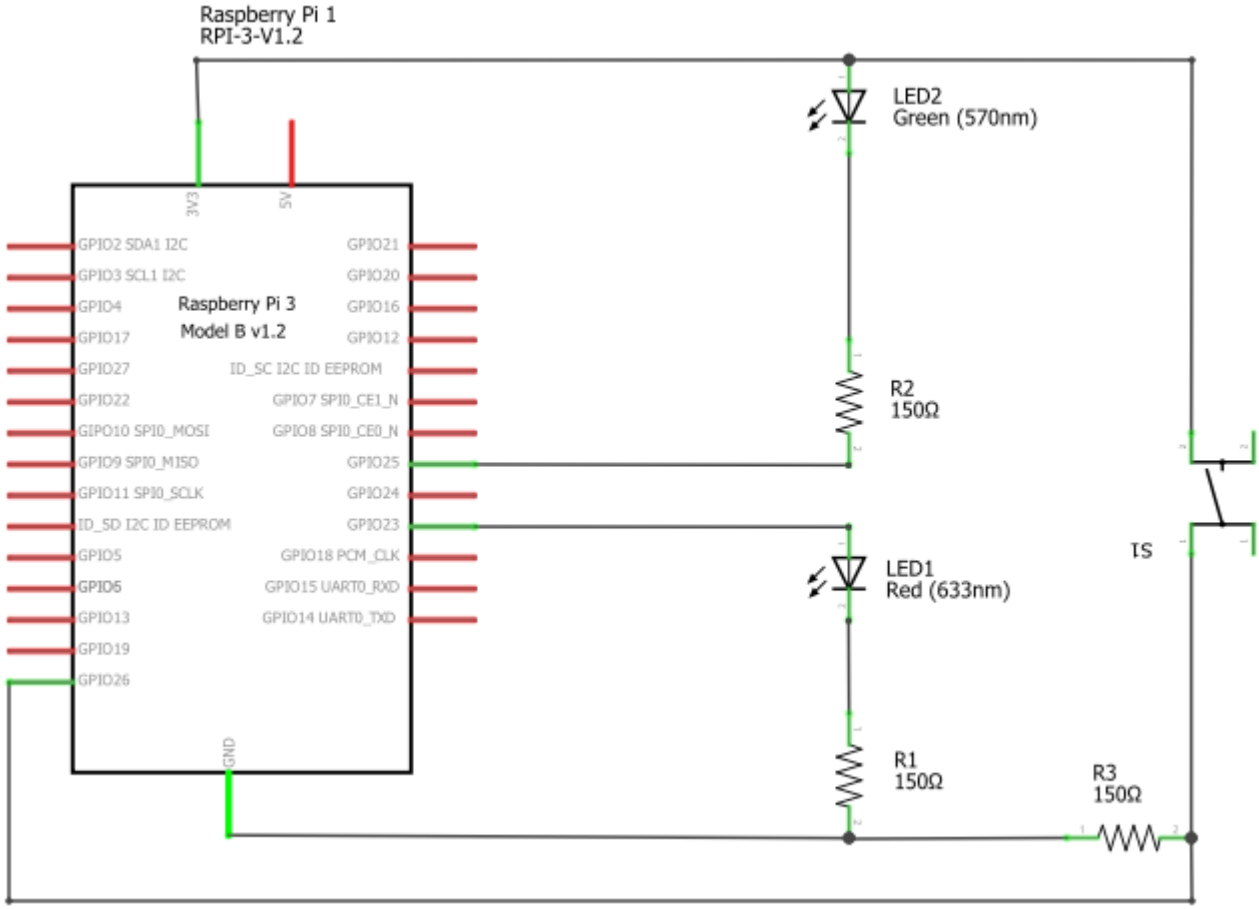


Fig.155

L'inserimento del pulsante tattile è immediato, come si vede dalla Fig.156, così come l'interfacciamento alla GPIO secondo le indicazioni dello schematics. Sono stati scelti cavi di collegamento del medesimo colore della Fig.154 così da rendere più semplice il cablaggio. La Fig.157 mostra il collegamento sulla GPIO.

La scrittura del codice C# per la gestione del pulsante, viene inserita come estensione dell'esempio precedente, quindi non è necessario creare una nuova applicazione, anche perché già in fase di design si è deciso di creare un'apposita Texbox che indichi l'effettiva pressione del pulsante. Tale codice deve ovviamente "reagire" alla pressione modificando in tempo reale il contenuto della "txtBottone".

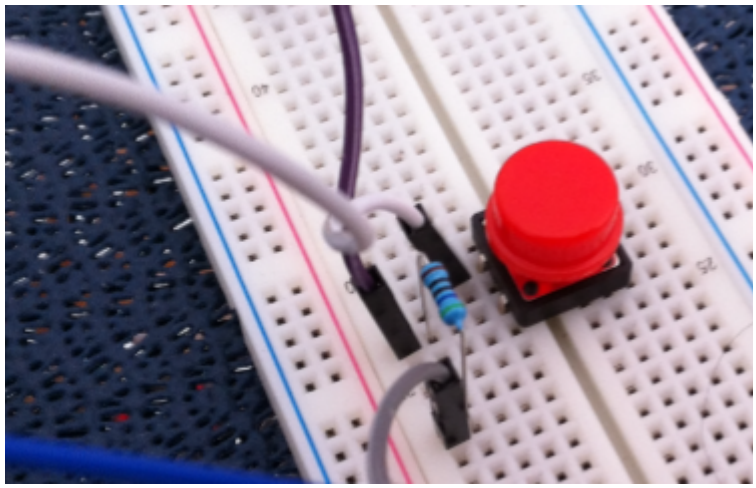


Fig.156

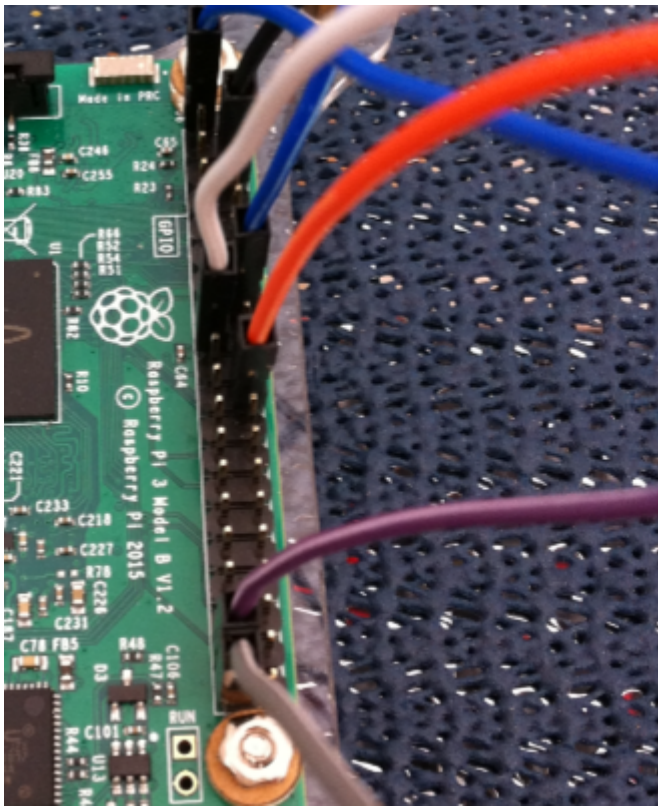


Fig.157

La riga di codice da inserire, senza eliminare nulla del precedente progetto, è la seguente:

```
using Windows.UI.Core;
```

Questa riga di codice, come spiegato in precedenza, serve per evitare di qualificare in modo esteso la classe "Windows.UI.Core" all'interno del programma. Il suo impiego è necessario perché la parte di codice che verifica la pressione del tasto è affidata a codice asincrono, quindi vi è un thread secondario che in caso di pressione dovrà notificarlo nella "txtBottone", la quale però è di proprietà dello "UI Thread". In altre parole vi è un interazione tra "UI Thread" che gestisce la grafica della UWP e il thread secondario che richiede una modifica su un componente grafico specifico. Questa interazione è legata ad un evento, ossia la pressione o meno del tasto, ed il contenuto del messaggio che si scambiano "UI Thread" e thread secondario è proprio la stringa di testo che si vuole stampare a video, stringa che verrà formattata come sequenza di caratteri unicode. Tutto il meccanismo di interazione, secondo quanto già esposto per il WPF, è possibile grazie alla presenza di una coda di messaggi asincrona a priorità chiamata dispatcher.
La politica di priorità della coda è "Normal" con la quale si gestiscono gli eventi nell'ordine in cui arrivano, quindi la coda a priorità viene utilizzata come coda "semplice". Per impostare questa politica di gestione di utilizza il membro "Normal" del tipo enumerativo "ClassDispatcherPriority" contenuto all'interno della classe "Windows.UI.Core".
All'interno della classe "MainPage" definire una costante intera che rappresenta la GPIO utilizzata secondo lo schematic di Fig.155. Il pin usato è la GPIO26 chiamato "pinDelBottone".

```
public sealed partial class MainPage : Page
{
    private const int pinLedRosso = 23;
    private const int pinLedVerde = 25;
    private const int pinDelBottone = 26;
     ...
}
```

All'interno del metodo "MainPage()" aggiungere il metodo di gestione del pulsante chiamato "gestisciPressioneTasto()".

```
    public MainPage()
    {
        this.InitializeComponent();
        AccendiLed();
```

```
            gestisciPressioneTasto();
        }
```

L'implementazione di questo metodo è molto semplice e per comodità di esposizione si sono indicati dei numeri in verde utili per commentare il significato della corrispondente riga di codice. Il metodo ha un ambito di visibilità di tipo privato e non restituisce nulla.

```
        private void gestisciPressioneTasto()
        {
            var gpio = GpioController.GetDefault();                    // 1)

            var pinBottone = gpio.OpenPin(pinDelBottone);              // 2)

            pinBottone.SetDriveMode(GpioPinDriveMode.Input);           // 3)

            pinBottone.DebounceTimeout = TimeSpan.FromMilliseconds(50); // 4)

            pinBottone.ValueChanged += valoreDelBottone;               // 5)
        }
```

La riga 1) alloca un handle alla GPIO, istruzione già vista in precedenza per la gestione del led che per comodità viene ri-dichiarata in questo blocco di codice. In un progetto software finalizzato alla vendita conviene dichiarare una ed una sola variabile "gpio", sfruttando magari il paradigma "IoC", qui si è preferito un'esposizione puramente didattica. Rispetto al contenuto del metodo "accendiLed()" non si è fatto il controllo se "gpio" è null, visto che tale test è appunto implementato il quel blocco di codice.
La riga 2) definisce un dato locale "pinBottone" che rappresenta la GPIO26 grazie al metodo "OpenPin()" al quale si passa la costante "pinDelBottone" precedentemente definita.
La riga 3) serve per definire il ruolo del pin del Raspberry. Rispetto alle maglie dei diodi led, il pin GPIO26 dovrà rilevare un livello di tensione e non fornirlo in uscita, motivo per cui il ruolo è in ingresso. Per l'impostazione si utilizza la costante "Input" della classe "GpioPinDriveMode" che vale 0. Tale costante va inserita nel metodo che imposta il ruolo ossia il "SetDriveMode()".
La riga 4) mette in evidenza un problema noto nell'ambito dei pulsanti, ossia il rimbalzo. Quando si preme il pulsante si vorrebbe che lo stato logico sia immediatamente alto, come

rappresentato in Fig.158, ma nell realtà il micro interruttore interno tende a rimbalzare, provocando un transitorio nel quale si susseguono lo stato logico alto e basso. La Fig.159 mostra l'aspetto reale del bouncing all'oscilloscopio.

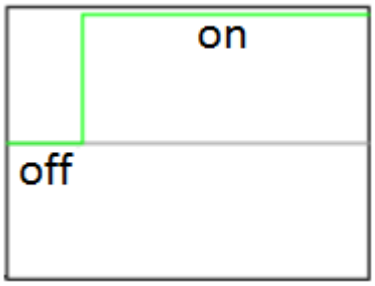


Fig.158

Fig.159

Per risolvere questa problematica senza utilizzare circuiti anti-bouncing, il linguaggio C# mette a disposizione l'attributo "DebounceTimeout" da applicare al solo GPIO26, quindi alla variabile "pinBottone". Il valore da assegnare a questo timeout rappresenta il tempo di attesa prima di leggere lo stato, di conseguenza è auspicabile che il transitorio sia già terminato con conseguente lettura dello stato corretto. Nel codice è stato impostato un lasso di tempo pari a 50ms sfruttando il metodo "FromMilliseconds" della classe "TimeSpan"

La riga 5) imposta il blocco di codice da chiamare quando si verifica l'evento relativo al cambio di stato rilevato sulla GPIO26. Questo codice è rappresentato dal metodo privato "valoreDelBottone" che ha il ruolo di callback, infatti nella riga non si sono utilizzate le parentesi tonde, oltre che impiegare i simboli "+=" ad indicare che esiste un evento delegate di sistema che inserisce in una lista tutte le callback da richiamare quando si verifica un determinato evento, in questo caso la pressione del tasto. Quando ciò accade l'unica callback che verrà chiamata sarà appunto la "valoreDelBottone". L'uso dei delegate

e delle relative callback è uno dei punti di forza dell'ambiente .NET, tecnica che nei linguaggi non gestiti come il C trova corrispondenza nei puntatori a funzione. Il codice della callback "valoreDelBottone" implementa il controllo dello stato del pulsante in modo asincrono grazie all'impiego di un thread secondario richiamato grazie al metodo "RunAsync" della classe Dispatcher.
La sintassi di questo metodo è la seguente:

```
public IAsyncAction RunAsync( CoreDispatcherPriority priority,
                              DispatchedHandler agileCallback)
```

Il punto di forza della RunAsync è proprio la sua natura asincrona, grazie alla quale il controllo torna immediatamente al chiamante così che possa venire eseguita l'istruzione successiva. Nel gergo dei sistemi operativi questa modalità è chiamata non bloccante. Il primo parametro che si deve passare alla RunAsync è la politica di priorità della coda, la quale come già indicato in precedenza sarà di tipo "Normal" ad indicare che si gestiscono gli eventi secondo una politica FIFO semplice senza quindi gestire le eventuali priorità. Il secondo parametro è la callback che deve eseguire il thread secondario, in questo caso la callback conterrà il codice per il controllo dello stato del pulsante notificandolo al componente "txtBottone". La RunAsync ritorna un handle che permette di capire se l'azione, e quindi il codice di controllo del pulsante, è terminato proprio perché essendo RunAsync asincrona, e ritornando il controllo al chiamante, non si ha più il controllo di cosa accade nel thread secondario. L'impiego di questa strategia a thread è di tipo thread-safe, visto che è già fornita dall'ambiente .NET, ma con il vincolo fondamentale per un corretto impiego è di eseguire parti di codice indipendenti tra di loro. Nel momento in cui i thread secondari devono accedere ad esempio alla stessa variabile o alla stessa struttura dati, tale costrutto di alto livello è inadatto perché possono sorgere dei problemi di sincronizzazione e quindi delle corse critiche. In questo caso il programmatore deve ricorrere a costrutti di sincronizzazione quali semafori, mutex, eventi e condition variable. Molto spesso nella definizione della RunAsync, si tende a non specificare il nome della callback, per poi realizzarla subito sotto sotto forma di metodo, ma si tende più ad impiegare le funzioni lambda tipiche dei linguaggi funzionali. Una funzione lambda è priva di nome e viene eseguita solo in quel contesto, riducendo quindi eventuali duplicazioni di nomi. Nel codice sotto si è espressamente voluto vedere un primo approccio che utilizza

esplicitamente la callback "miaCallBack" a titolo puramente didattico, quindi non è da inserire all'interno del codice del progetto.

```
private void valoreDelBottone(GpioPin sender, GpioPinValueChangedEventArgs e)
{
  var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, miaCallback);
  ...
}

private void miaCallback()
{
   ...
}
```

La sintassi in C# di una funzione lambda è la seguente:

(< parametri >) => { <istruzioni>; }

Sotto è riportato un esempio per cercare di capire la logica di funzionamento di una funzione lambda così che poi si capisca in modo corretto il suo impiego nel codice del progetto.

```
 public void esempioLambda(int dato)
{
  int a=0;              Lambda function
  bool test =  () => { a=20; return a>dato; }
  Console.WriteLine(test);
}

...
esempioLambda(30);  // L1)
esempioLambda(5);   // L2)
```

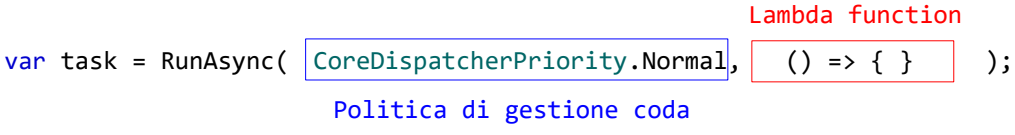
La riga L1) chiama il metodo "esempioLambda" a qui si passa per copia lo scalare 30 nello stack. Questo metodo poi alloca, sempre nello stack, un dato locale intero "a=0" e successivamente anche un dato booleano chiamato "test". In questo momento viene definita una funzione lambda a cui non vengono passati parametri dall'esterno (le

parentesi tonde sono vuote), ma viene modificato per riferimento il valore della variabile "a=20", questo sta a significare che il CLR ha creato nello HEAP una locazione di memoria rappresentata ora dalla variabile riferimento "a" il cui contenuto è pari a 20. Questo aspetto è fondamentale per capire ed usare le funzioni lambda in C# , infatti se si utilizzasse il linguaggio C++, la variabile "a" non sarebbe mai trattata come riferimento, ma semplicemente come dato locale. La funzione lambda di Fig.161 effettua il controllo tra il dato passato "30" e il contenuto "a=20", restituendo all'interno "false" in "test". La "Console.WriteLine" non fa altro che stampare su linea di comando la scritta "false." Ripetendo il metodo con lo scalare "5", riga L2, si ottiene "true".
Il secondo esempio permette di capire la logica dei parametri passati dall'esterno alla funzione lambda.

```
delegate bool callback(int dato);                    // L1)
callback cb;                                         // L2)

class Esempio
{
  public void esempioLambda2(int dato)
  {
    int a=0;              Lambda functions
    bool test =  () => { a=20; return a>dato; }      // L3)
    cb =   (parametro) => { return parametro==a; }   // L4)
    Console.WriteLine(test);
  }
}

public static void Main()
{
  Esempio e=new Esempio();
  e.esempioLambda2(30);
  bool test = e.cb(20);                              // L5)
  Console.WriteLine(test);
}
```

La riga L1) dichiara un delegate chiamato "callback" di tipo "bool". Il delegate è un tipo di dato che permette la gestione del pattern callback.
La riga L2) dichiara una variabile "cb" di tipo callback.

La riga L3) definisce la funzione lambda vista nell'esercizio precedente, quindi la variabile locale "test" sarà uguale a "false" visto che la variabile di istanza "e" passa al metodo "esempioLambda2()" lo scalare "30" che è diverso da "a=20".
La riga L4) è molto interessante visto che la funzione lambda viene assegnata ad una variabile di tipo delegate, il che significa che si procede alla registrazione della funzione lambda, ma non alla sua esecuzione. Un programmatore C direbbe che "cb" è un puntatore a funzione, ossia una variabile che contiene l'indirizzo di una funzione e che la registrazione significa proprio specificare nella variabile "cb" l'indirizzo di una data funzione che contiene il codice da eseguire. Nel mondo .NET il puntatore a funzione è rappresentato da una variabile di tipo delegate che invoca una callback nel momento in cui si verifica un dato evento. La riga L4) registra la callback, ossia registra la funzione lambda, ma non la esegue perché sarebbe necessario invocare il delegate passando un parametro visto che questa seconda funzione lambda specifica un parametro chiamato "parametro". L'istruzione "Console.WriteLine()" successiva stamperà "false" in riferimento alla riga L3).
La riga L5) chiama la variabile delegate alla quale si passa lo scalare "20". Questa invocazione non fa altro che chiamare la callback associata, ossia la funzione lambda, eseguendone il codice che ritorna "true" perché "parametro" è stato passato per copia e vale "20", la variabile "a" è un riferimento nello HEAP che vale anch'essa 20, quindi il confronto non può che restituire un valore "true".
Sovente si utilizzano le chiamate a variabili delegate in eventi di sistema così che si possano eseguire le istruzioni presenti nelle callback, previa registrazione secondo la filosofia della riga L4).
Dopo questa breve panoramica è possibile riassumere il metodo RunAsync definito con la funzione lambda da impiegare nel progetto nel seguente modo:



Segue sotto il codice completo della callback "valoreDelBottone" da inserire nel progetto.

```
private void valoreDelBottone(GpioPin sender, GpioPinValueChangedEventArgs e)
{
    var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
                                        {
                                          if (e.Edge == GpioPinEdge.RisingEdge)
                                              txtBottone.Text = "Bottone premuto!";
                                          else
                                              txtBottone.Text = "Bottone rilasciato!";
                                        }
                                    );
}
```

Il corpo della funzione lambda contiene un semplice controllo condizionale che verifica se vi è stato passaggio dallo stato logico basso a quello alto grazie al valore di confronto determinato dal membro "RisingEdge" pari a "1" del tipo enumerativo "GpioPinEdge". Il parametro "e" è una classe che contiene i dettagli inerenti la GPIO26, quindi con il metodo "Edge" si verifica il tipo di cambiamento che è avvenuto su questo pin. Nel caso il cambiamento corrisponda al passaggio dallo stato basso e quello alto allora si stampa il messaggio "Bottone premuto!" nella "txtBottone", in caso contrario si riporta la scritta "Bottone rilasciato!". La callback "valoreDelBottone" gira in modalità asincrona grazie ad un thread secondario, ed ogni volta che si preme il tasto viene invocata così che si possa modificare correttamente il valore di "txtBottone". Tutto questo è stato possibile visto che nel metodo "gestisciPressioneTasto()" si è provveduto alla registrazione della callback "valoreDelBottone".

Terminata la scrittura del codice è necessario procedere al deployment e alla sua esecuzione. Il risultato a livello di bread board è quello di Fig.160.

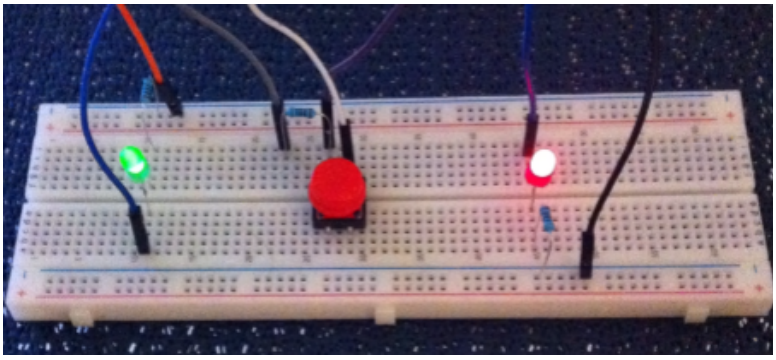


Fig.160

La UWP visualizzerà sul monitor nella "txtBottone" il messaggio di inizializzazione, per poi stampare, sulla base o meno della pressione del tasto, il messaggio "Tasto premuto!" oppure "Tasto rilasciato!" come si vede nelle Fig.161, Fig.162 e Fig.163.

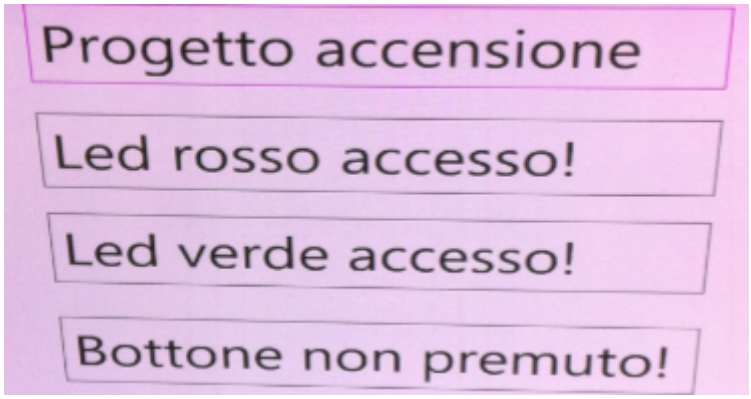


Fig.161

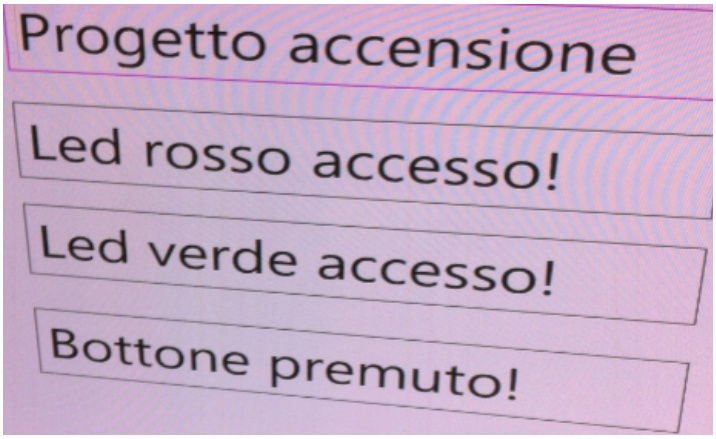


Fig.162

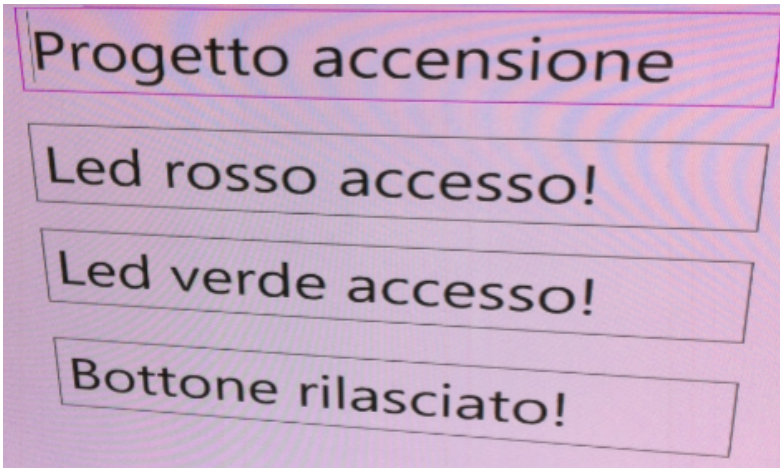


Fig.163

3.2.7 Codice sorgente C# completo per gestione led e pulsante

Un test significativo è quello di cambiare i pin GPIO utilizzati per i led e per il pulsante così da testare il corretto funzionamento di tutta la GPIO sia in entrata che in uscita. Per completezza si riporta tutto il codice C# di questo progetto di testing.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
using Windows.Devices.Gpio;
using Windows.UI.Core;


namespace AccensioneLed
{
    public sealed partial class MainPage : Page
    {
        private const int pinLedRosso = 23;
        private const int pinLedVerde = 25;
        private const int pinDelBottone = 26;

        public MainPage()
        {
            this.InitializeComponent();
            AccendiLed();
            gestisciPressioneTasto();
        }
```

```
    private void gestisciPressioneTasto()
    {
        var gpio = GpioController.GetDefault();

        var pinBottone = gpio.OpenPin(pinDelBottone);

        pinBottone.SetDriveMode(GpioPinDriveMode.Input);

        pinBottone.DebounceTimeout = TimeSpan.FromMilliseconds(50);

        pinBottone.ValueChanged += valoreDelBottone;
    }

    private void valoreDelBottone(GpioPin sender, GpioPinValueChangedEventArgs e)
    {
        var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
                                            {
                                              if (e.Edge == GpioPinEdge.RisingEdge)
                                                  txtBottone.Text = "Bottone premuto!";
                                              else
                                                  txtBottone.Text = "Bottone rilasciato!";
                                            }
                                        );
    }

private void AccendiLed()
{
        var gpio = GpioController.GetDefault();

        if(gpio==null)
        {
            txtBenvenuto.Text = "Device senza GPIO!";

            return;
        }
        else
        {
            var pinRosso = gpio.OpenPin(pinLedRosso);
            pinRosso.Write(GpioPinValue.High);
            pinRosso.SetDriveMode(GpioPinDriveMode.Output);
```

```
                txtLedRosso.Text = "Led rosso accesso!";

                var pinVerde = gpio.OpenPin(pinLedVerde);
                pinVerde.Write(GpioPinValue.Low);
                pinVerde.SetDriveMode(GpioPinDriveMode.Output);
                txtLedVerde.Text = "Led verde accesso!";
            }
        }
    }
}
```

Capitolo 4

Stazione meteo del vento

Questo capitolo descrive la realizzazione di una stazione meteo del vento tramite un anemometro La Crosse TX20, evidenziando le fasi di progettazione hardware e software.

4.1 Anemometri

La scelta dell'anemometro è caduta sul La Crosse TX20 essendo un prodotto molto semplice da utilizzare e dai costi contenuti. Assieme al modello TX20 è stato testato anche il TX23U, il quale si differenzia per la gestione dell'invio dei dati.

4.1.1 La Crosse TX20

Questo modello di anemometro ha la caratteristica fondamentale di inviare in modo continuo ogni 2 secondi i dati tramite un protocollo di comunicazione riservato che, tramite operazioni di reverse engineering, è stato correttamente interpretato. Verranno quindi utilizzate le informazioni riportate sui forum in rete, così da evitare di rifare tutta l'analisi con conseguente risparmio di tempo. La Fig.164 mostra il modello TX20 e come si vede nella confezione di vendita viene fornita una torretta in plastica e dei supporti a T così da permettere vari tipi di installazione. Il dispositivo è collegato ad un cavo 4 fili di lunghezza 10m con plug RJ11 per la connessione alla base La Crosse non utilizzata in questo progetto, visto che sarà il Raspberry Pi 3 a fungere da stazione ricevente.

Fig.164

La frequenza di trasmissione dei dati è di 433MHz, mentre la distanza anemometro-base può essere al massimo 100m. Il plug di connessione RJ11 ha una crimpatura secondo le indicazione della Tab.9 così come si vede chiaramente dalla Fig.165.

Pin	Colore	Significato
1	giallo	Massa dell'anemometro
2	verde	Segnale DTR di avvio trasmissione. Metterlo a massa per impostare già la trasmissione dei dati ogni 2 secondi.
3	rosso	Alimentazione dell'anemometro di +3.3V
4	nero	Segnale TX di trasmissione dati

Tab.9

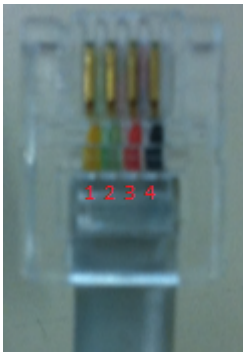


Fig.165

Durante i test di laboratorio, al fine di definire la Tab.9, si è proceduto all'apertura del TX20 e del TX23U così da verificare i collegamenti sulla scheda interna, come si vede chiaramente nelle immagini Fig.166 (TX23U) e Fig.167.

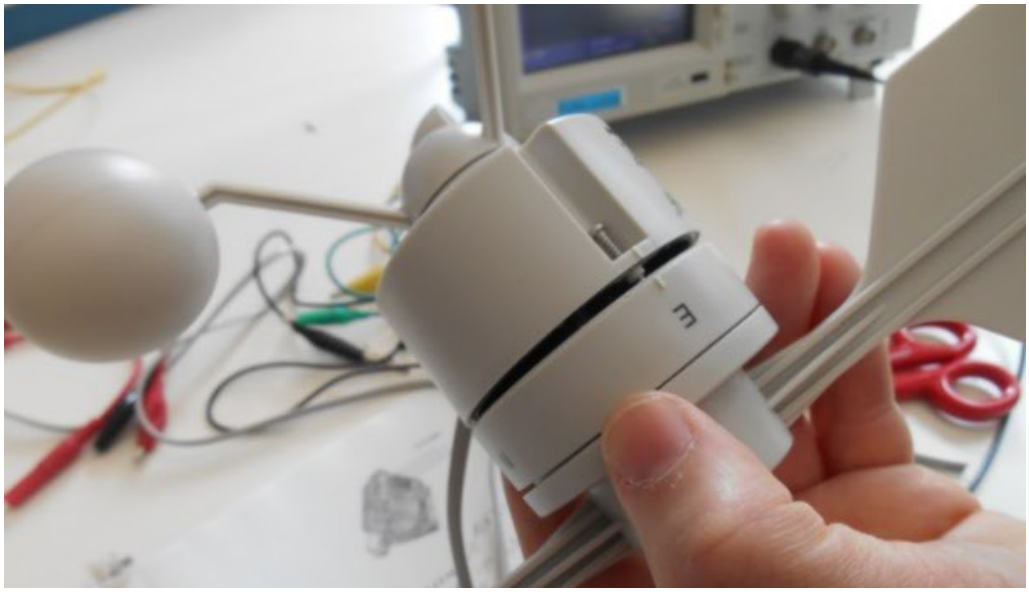


Fig.166

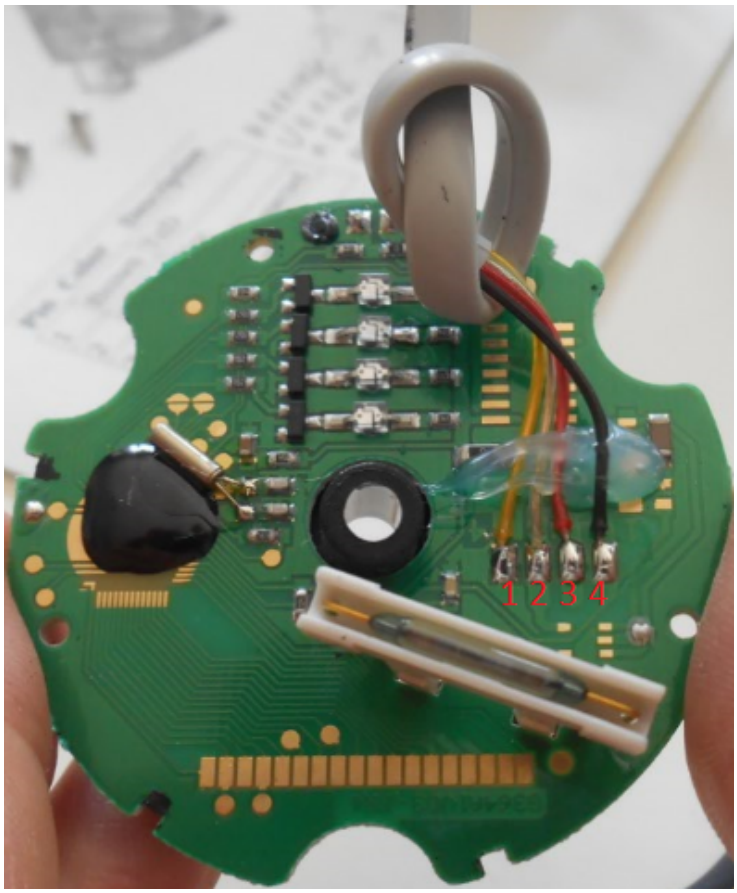


Fig.167

Come indicato nella Tab.9, per richiedere l'inizio della trasmissione dei dati dal TX20, è necessario che sul DTR di colore verde vi sia lo stato logico basso, quindi conviene collegare tale cavo ad un pin della GPIO del Pi e a livello di codice impostare in uscita il livello di tensione basso per iniziare a leggere i dati. Al termine della lettura conviene poi rilasciare le risorse porte GPIO così che il TX20 cessi la trasmissione. E' ragionevole ricevere i dati ogni 1-2 secondi, quindi si andrà ad impiegare un timer a livello di C#. Il TX20 è stato costruito perché spedisca in modo continuo, con un intervallo di 2 secondi, i dati nel momento in cui il DTR è posto a massa. La Fig.168 riporta la tipologia di connessione che è estremamente banale. Per comodità viene rimosso il plug RJ11, così che l'alimentazione di +3.3V del Pi venga connessa al cavo rosso, la massa GND al cavo giallo, la GPIO4 al cavo nero col fine di trasmettere i dati e la GPIO17 al cavo verde DTR tramite il quale si avvia la trasmissione. Qualora sia in dotazione un RJ11 femmina è possibile evitare di tagliare il plug RJ11, anche se un nuovo crimpaggio è un'operazione molto semplice se si ha in dotazione la pinza ed un nuovo plug. Ovviamente il nuovo crimpaggio deve seguire la pinatura di Fig.165.

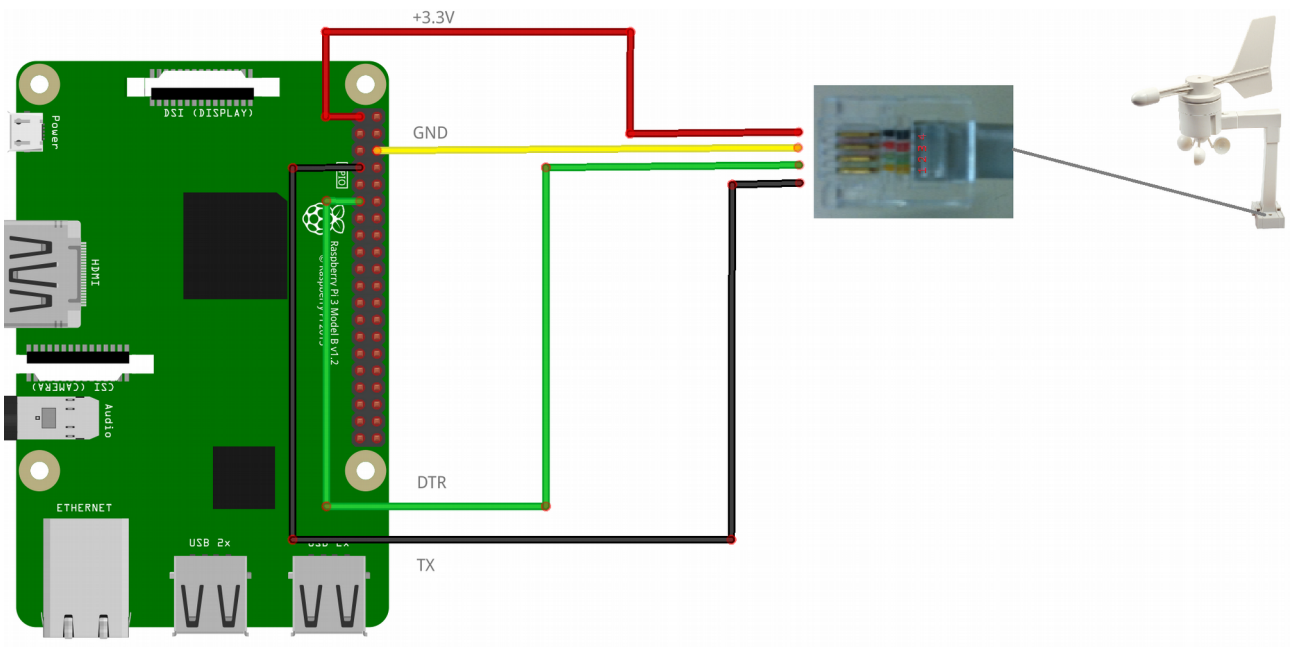


Fig.168

4.1.2 La Crosse TX23U

Questo modello di anemometro ha la caratteristica fondamentale che non invia nulla in modo continuo, ma solo un datagramma previa richiesta della base. Il TX20 invece, previo livello logico basso sul DTR, trasmette datagrammi in modo continuo ogni 2 secondi. La tipologia di connessione segue la medesima pinatura e vale sia la Tab.9 che la Fig.165. Quello che cambia da un punto di vista è l'inserimento di una resistenza di pull-up tra l'alimentazione e il segnate TX di trasmissione come si vede dalla Fig.169 e dal relativo schematic di Fig.170.

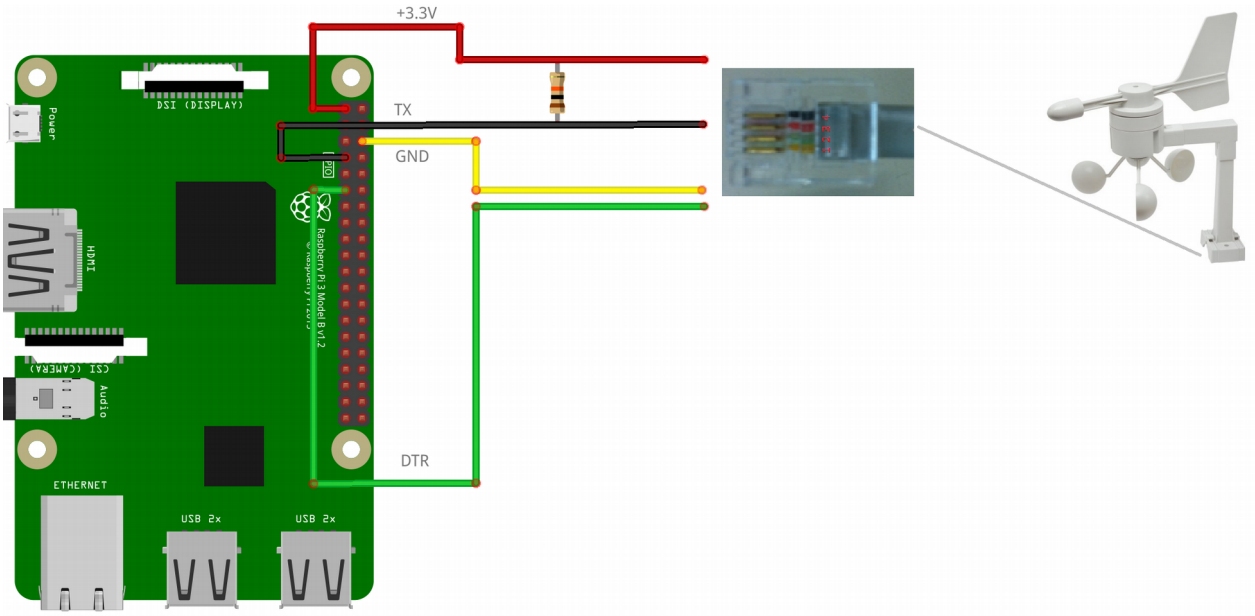


Fig.169

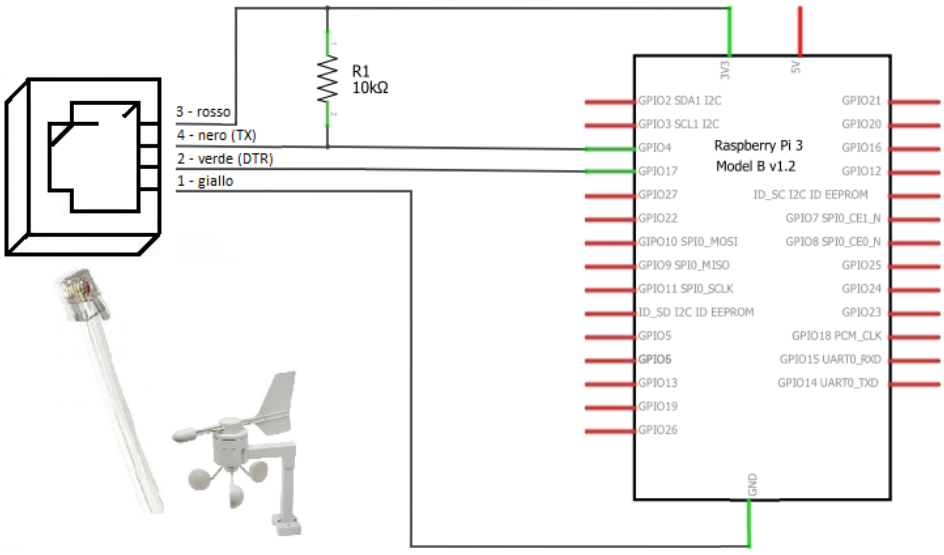


Fig.170

4.1.3 Protocollo TX20 e TX23U

Gli anemometri della La Crosse sono prodotti commerciali soggetti a diritti di autore, quindi il protocollo di comunicazione utilizzato per inviare i dati alla base non è di libero accesso. Il sito web https://www.john.geek.nz/2011/07/la-crosse-tx20-anemometer-communication-protocol/ di John Burns elenca in dettaglio il procedimento di reverse engineering che ha prodotto in modo accurato il datagramma della trasmissione. La Fig.171 mostra un esempio dei treni di impulsi con le varie sezioni relative ad informazioni diverse.

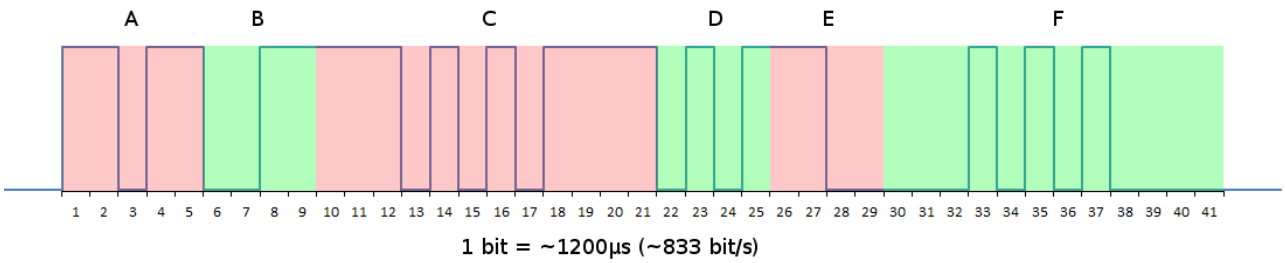


Fig.171

Il datagramma è composto da 41 bits divisi in 6 sezioni. Ogni bit ha una lunghezza di 1220µsec. La Tab.10 riporta il significato delle varie sezioni.

Sezione	N° bits	Usa logica invertita	Endianess	Descrizione
A	5	Si	LSB	Start frame 11011
B	4	Si	LSB	Direzione del vento
C	12	Si	LSB	Velocità del vento
D	4	Si	LSB	Checksum
E	4	No	LSB	Direzione del vento
F	12	No	LSB	Velocità del vento

Tab.10

Buona parte dei bits del datagramma utilizzano la logica inversa ossia 1 logico è pari a 0V mentre 0 logico è pari a +3.3V. E' possibile inserire un circuito che effettua l'inversione, ma non essendo la logica invertita applicata a tutti i bits del datagramma è da valutare l'applicazione della relativa circuiteria. In Fig.172 è riportato il circuito elettrico utilizzato.

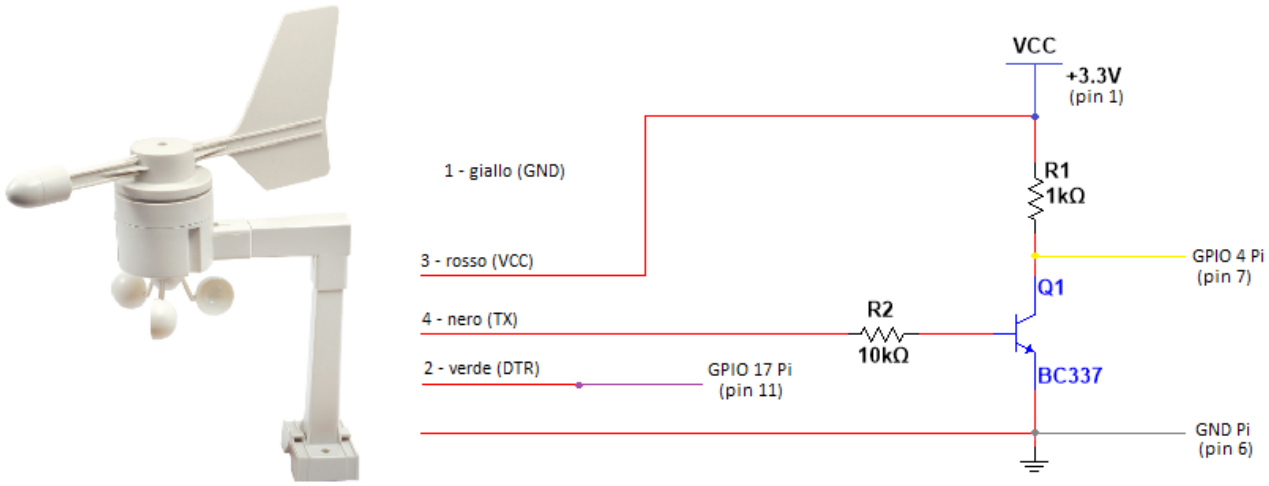


Fig.172

Inserendo un BJT BC337 è possibile farlo lavorare come interruttore, ribaltando la logica inversa del datagramma, visto che nel momento in cui TX è a 1 logico, ossia a +3.3V, il transistor va in conduzione è la corrente di collettore va a massa ponendo il pin GPIO4 a potenziale basso. Quando il il TX è a 0 logico, ossia 0V, il transistor non conduce, quindi il potenziale della GPIO4 è pari a quella di VCC ossia +3.3V e quindi 1 logico. Nella realizzazione del progetto viene utilizzata la circuiteria di Fig.172 nella quale sono state colorate le connessioni con i colori dei cavi che si connettono ai rispettivi pin del Pi.

Prima di testare il funzionamento fisico dell'anemometro, è obbligatorio analizzare la sequenza dei bits che compongono il datagramma di Fig.171 come da Tab.10.

I primi 5 bits servono per la sincronizzazione della trasmissione, così che il Pi possa poi, grazie al codice C#, acquisire in modo corretto i bit successivi interpretandoli in modo adeguato. La sequenza di questo start frame è 11011, sempre in logica invertita. Ogni bit impiega 1220µsec quindi, tra un bit e l'altro, è sempre necessario inserire un ritardo preciso, onde evitare di avere un accumulo di bits letti nel momento errato che portano, di conseguenza, a dati non significativi di velocità e direzione del vento.

E' importante ricordare che il TX20 inizia la trasmissione solo nel momento in cui il DTR viene posto a livello logico basso, motivo per cui si deve intervenire al livello di codice C# per impostare sul GPIO17 una tensione bassa prima di attendere lo start frame. Alla fine dell'analisi del datagramma si deve rilasciare la GPIO17 così che il programma possa, trascorso un lasso di tempo del timer, ripetere nuovamente il codice riabilitando lo stato basso sulla GPIO17 in attesa nuovamente dello start frame. La logica si ripete ciclicamente.

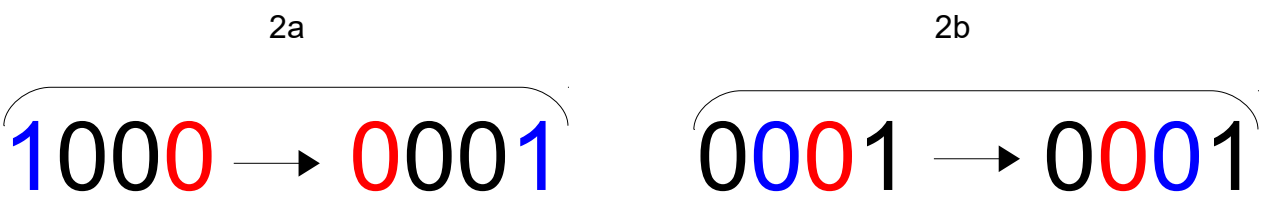
I successivi 4 bits indicano la direzione del vento, quindi le possibilità sono al massimo 2^4=16. Una volta ricevuti tutti e quattro i bits è necessario provvedere alla loro inversione (se non si è fatta inversione hardware), ma non è possibile estrarre subito la direzione perché la logica di trasmissione è di tipo endianess, ossia si provvede prima a trasmettere il bit meno significativo LSB e poi i vari bits più significativi. Un esempio pratico chiarisce la logica di funzionamento. Si supponga di ricevere i seguenti bits:

<direzione-vento-trasmessa-su-TX> = 0111

Fase 1: Inversione logica software o hardware

<direzione-vento-inversione-BJT> = 1000

Fase 2: Scambio tra i bits LSB (blu) e i bits MSB (rosso)



Prima si procede ad invertire le coppie di bits più esterne "1--0" così da ottenere il risultato finale "0--1" della prima fase 2a), poi si procedere alla coppia di bits interna "-00-" della fase 2b) che produce ovviamente "-00-". La logica va applicata a tutti i bits partendo sempre dall'esterno verso l'interno.

<direzione-vento-corretta> = 0001

Resta ancora da capire il numero binario 0001 a quale direzione corrisponde. La Tab.11 riporta le direzioni relative alla sequenze di bit corrette.

Valore binario corretto	Direzione
0000	Nord (N)
0001	Nord Nord Est (NNE)
0010	Nord Est (NE)
0011	Est Nord Est (ENE)
0100	Est (E)
0101	Est Sud Est (ESE)
0110	Sud Est (SE)
0111	Sud Sud Est (SSE)
1000	Sud (S)
1001	Sud Sud Ovest (SSW)
1010	Sud Ovest (SW)
1011	Ovest Sud Ovest (WSW)
1100	Ovest (W)
1101	Ovest Nord Ovest (WNW)
1110	Nord Ovest (NW)
1111	Nord Nord Ovest (NNW)

Tab.11

L'esempio precedente trova quindi corrispondenza con la direzione del vento NNE. Considerando le 16 combinazioni massime, ed avendo l'angolo giro 360°, è possibile considerare 1/16 di rivoluzione partendo da nord (0000), quindi sulla base della Tab.11, la direzione NNE corrisponde proprio a 360°/16 = 22.5°. Qualora la direzione fosse ENE, la rivoluzione sarebbe pari a 22.5°*3=67.5°.

I successivi 12 bits indicano la velocità del vento in m/s e il valore decimale corrispondente deve venire moltiplicato per lo scalare 0.1m/s così da ottenere la velocità effettiva sempre in m/s. Qualora dalla conversione da binario a decimale si ottiene 100, moltiplicando per 0.1m/s, si ottiene una velocità di 10m/s che corrisponde a 36km/h (si moltiplica per 3.6) e a 22 miglia orarie (si moltiplica per 2.2). Dei bits solo i primi 9 portano informazione, mentre i 3 bits più significativi sono posti sempre a 000, di conseguenza il massimo valore decimale è 2^9-1=511, che significa 511*0.1m/s ossia 51.1m/s pari a 183.96km/h (si moltiplica per 3.6). La sequenza dei bits è sempre a logica invertita, quindi dopo aver negato i bits, si devono scambiare gli LSB e MSB. Segue un esempio.

<velocità-vento-trasmessa-su-TX> = 111010101111

Fase 1: Inversione logica software o hardware

<direzione-vento-inversione-BJT> = 000101010000

Fase 2: Scambio tra i bits LSB (blu) e i bits MSB (rosso)

<velocità-vento-corretta> = 000010101000
<velocità-vento-corretta-su-9bits> = 010101000

I primi 3 bits MSB non vanno considerati quindi il numero binario corretto per elaborare la velocità è 010101000 che corrisponde al numero decimale 168, scalare che va moltiplicato per 0.1m/s da cui ne consegue una velocità di 16.8m/s pari a 60.48km/h.
I successivi 4 bits rappresentano il checksum, ossia una sequenza di bits che permettono di verificare la correttezza della direzione e della velocità del vento. La logica di trasmissione è sempre invertita così come la trasmissione iniziale dei bits LSB. Il punto cardine è come verificare, tramite il checksum, l'integrità dei dati, visto che la trasmissione

potrebbe subire interferenze producendo dati errati. Il checksum ricevuto deve venire comparato con una somma binaria ottenuta nel seguente modo:

<direzione-vento-corretta>: <4 bits>
<velocità-vento-corretta>: <bit da 1 a 4>
<velocità-vento-corretta>: <bit da 5 a 8>
<velocità-vento-corretta>: <bit da 9 a 12>

Risultato: <SOMMA bit a bit> = <checksum-corretto> ?

Il risultato della somma deve essere identico al checksum, il quale prima deve essere invertito e successivamente risistemato in ottica MSB e LSB. Identico discorso vale per la velocità e per la direzione che devono già essere sistemate come svolto in precedenza. Qualora la somma sia diversa del checksum, significa che i dati non sono integri e devono venire scartati. Un esempio è utile per chiarire la dinamica dell'utilità del checksum.

<direzione-vento-corretta> = 0001
<velocità-vento-corretta> = 000010101000
<checksum-su-TX> = 0011

La direzione e la velocità del vento sono già state sistemate in termini di inversione e bits LSB e MSB, non appunto vengono riportati per comodità i due esempi precedenti. Il checksum risulta essere quello spedito dal TX20, quindi si deve procedere all'inversione di logica, che porta ad avere la sequenza 1100. Lo step successivo è applicare lo scambio tra i bit LSB e MSB come fatto per direzione e velocità del vento.

<checksum- corretto> = 0011

Determinato il corretto valore del checksum, si procede alla somma bit a bit considerando tre gruppi di 4 bits per la velocità del vento come indicato in precedenza.

<direzione-vento-corretta> = 0001
<velocità-vento-corretta> = 000010101000
<checksum-corretto> = 0011

bits della direzione vento: 0 0 0 1 +
bits da 1 a 4 velocità vento: 0 0 0 0 +
bit da 5 a 8 velocità vento: 1 0 1 0 +
bits da 9 a 12 velocità vento: 1 0 0 0 =
---
risultato della somma: 0 0 1 1

Risultato della somma e checksum sono identici, quindi direzione e velocità sono in apparenza integri. I successivi 4 bits, rappresentano ancora la direzione del vento, solo che vengono trasmessi già invertiti, quindi è necessario effettuare il complemento tramite il codice C#, qualora via sia l'inversione hardware tramite BJT, inoltre è sempre richiesto lo scambio tra i bits LSB e MSB. Questo valore è un duplicato di quello inviato invertito, quindi oltre al checksum, è necessario che i due valori siano identici. Lo stesso discorso accade per gli ultimi 12 bits che sono un doppione delle velocità del vento, sequenza che richiede anch'essa il complemento a livello software ed il successivo scambio tra i bit LSB e MSB. I due valori delle velocità devono obbligatoriamente essere identici, per avere la certezza che la trasmissione sia completamente integra, visto che lo stesso checksum potrebbe subire interferenze. Le fasi operative dell'analisi del datagramma riassunte sono:

1. Lettura dello start frame 11011 in logica invertita
2. Lettura dei 4 bits della direzione del vento, inversione e scambio LSB con MSB
3. Lettura dei 12 bits della velocità del vento, inversione e scambio LSB con MSB
4. Lettura dei 4 bits del checksum, inversione e scambio LSB con MSB
5. Somma binaria della direzione e dei 3 campi da 4 bits della velocità del vento
6. Comparazione somma binaria e checksum
7. Se la somma collide con i checksum i dati sono in apparenza integri
8. Lettura dei 4 bits della direzione del vento, complemento e scambio LSB con MSB
9. Lettura dei 12 bits della velocità del vento, complemento e scambio LSB con MSB
10. Confronto primo valore e secondo valore di direzione del vento

11. Qualora i due valori collidono la direzione del vento è integra
12. Confronto primo valore e secondo valore della velocità del vento
13. Qualora i due valori collidono la velocità del vento è integra

Il montaggio su breadboard è molto semplice, come si evince dalla Fig.173.

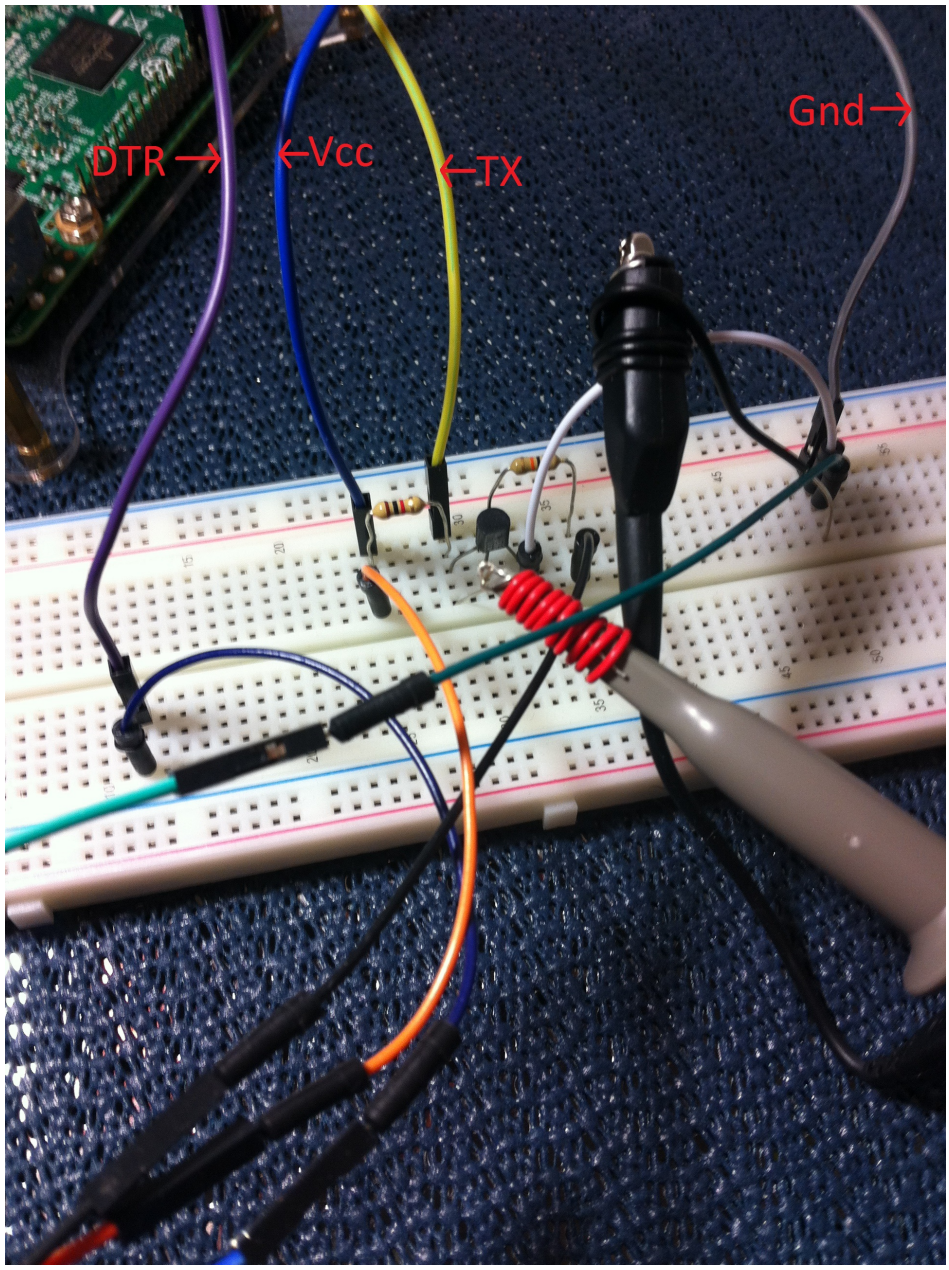


Fig.173

Per comodità sono stati indicati sui cavi diretti al Pi la tipologia di segnale nel rispetto di Fig.172

La Fig.174 riporta lo schema elettrico con Fritzing, mentre la Fig.175 riporta lo schematics.

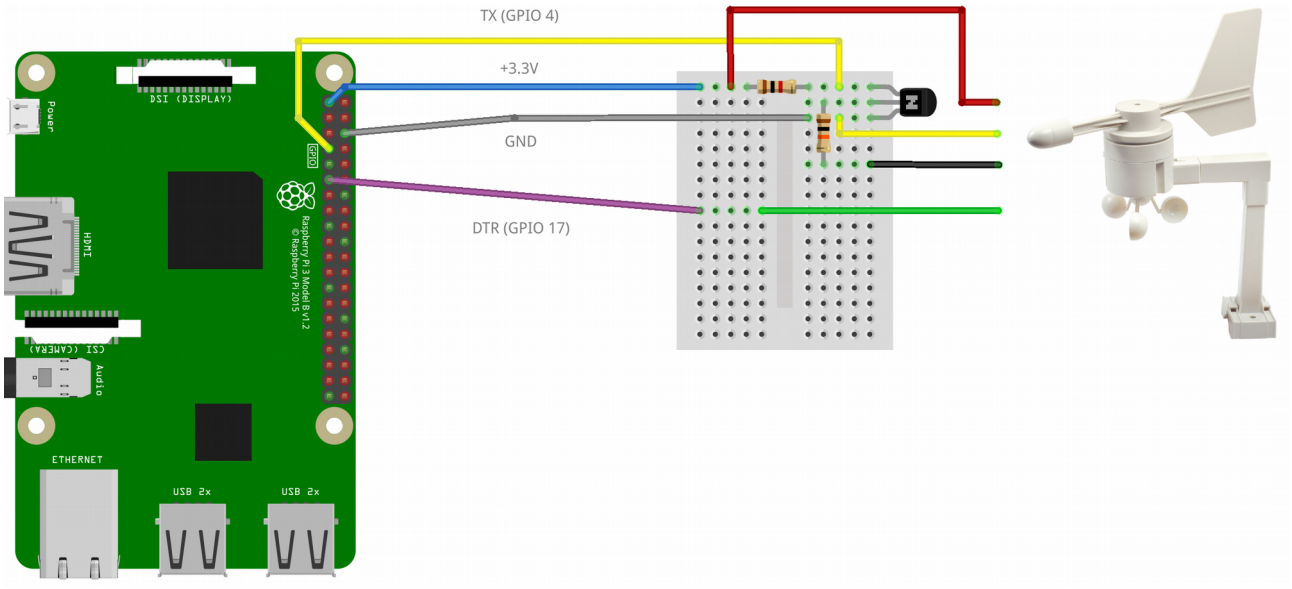


Fig.174

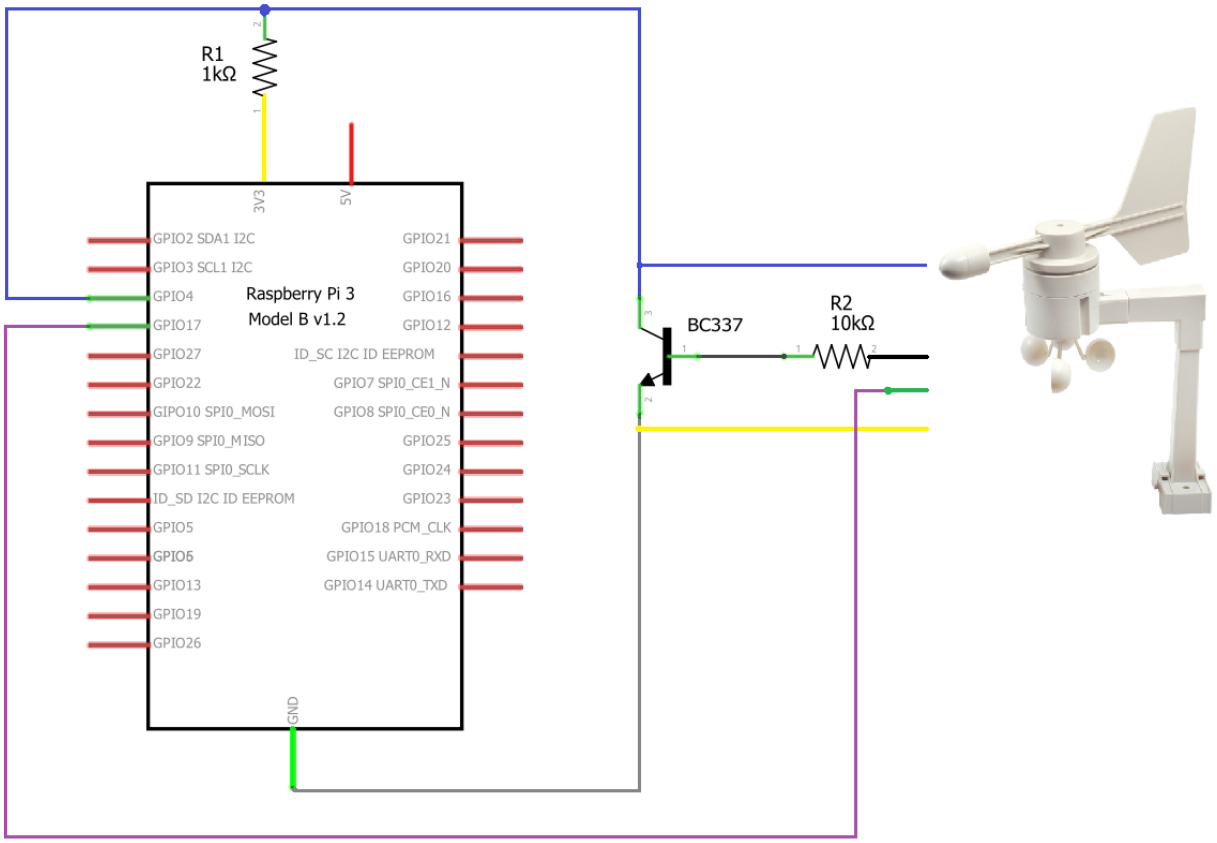


Fig.175

Terminato il cablaggio fisico, non resta che alimentare il Raspberry Pi 3 e verificare, tramite un oscilloscopio digitale DSO, la presenza o meno di qualche forma d'onda sul collettore del BJT, dove sarà sempre presente il segnate TX invertito. E' molto importante sottolineare che fino a questo momento nessun codice di programmazione è in esecuzione sul sistema, visto che nessuna riga di C# è stata scritta, quindi è presumibile, essendo il DTR collegato del GPIO17 non programmato, l'assenza di qualsiasi datagramma. Per riassumere in due parole l'anemometro non dovrebbe trasmettere.

Il DSO utilizzato è un Rigol DS1052E 50MHz 1GSa/s, dispositivo più che adatto per analizzare un datagramma di quasi 50msec.

Il collegamento prevede di utilizzare una sonda di compensazione 1X/10X collegata al collettore del BJT direttamente al CH1, come si vede chiaramente dalla Fig.173 con il filo rosso ancorato all'innesto della sonda. Ovviamente la massa della sonda va collegata alla massa dell'intero circuito così da avere una maglia equipotenziale. Per semplificare l'analisi del segnale, il DSO è stato collegato al calcolatore tramite USB così da visualizzare l'onda direttamente sul software UltraScope, tramite il quale è possibile controllare remotamente l'oscilloscopio. Sorprendentemente l'oscilloscopio rileva un datagramma, come si evince dalle Fig.176 e Fig.177.

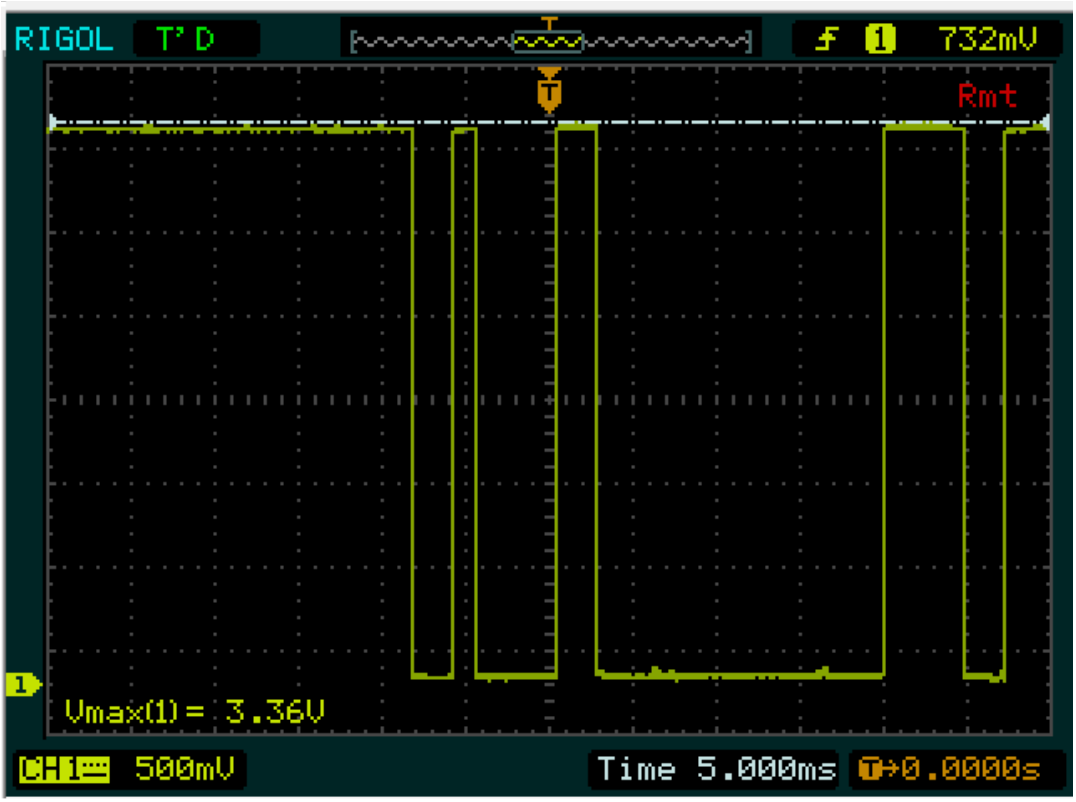


Fig.176

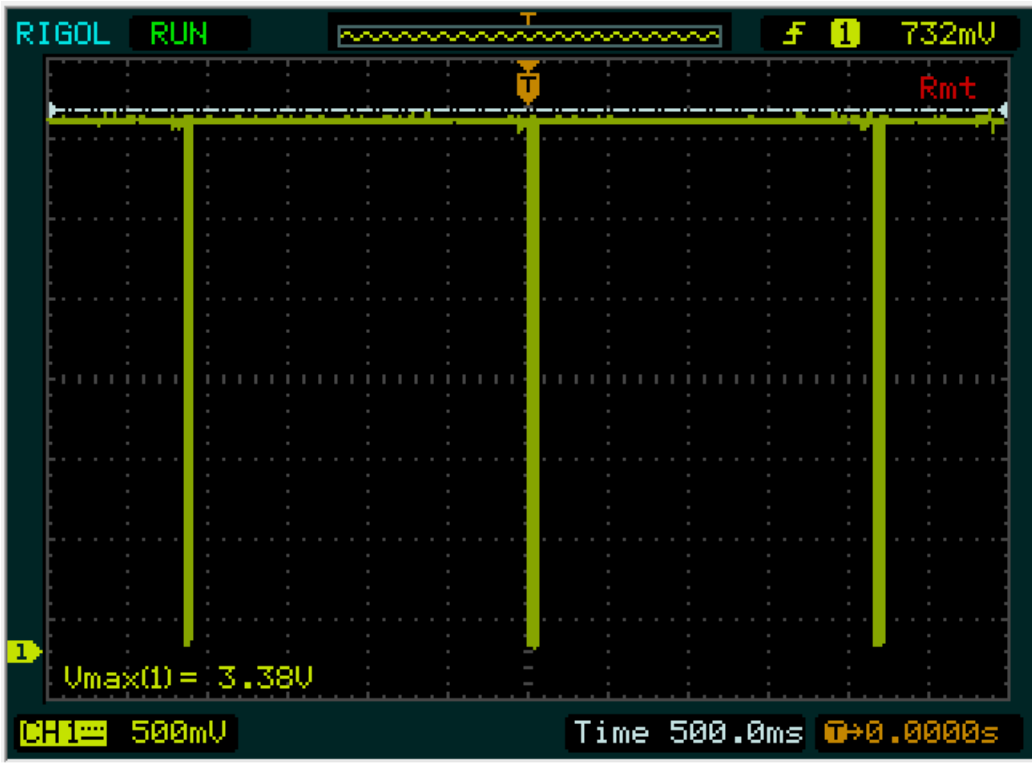


Fig.177

In quest'ultima immagine, con un tempo di divisione sull'asse X di mezzo secondo, si capisce chiaramente che il datagramma viene inviato ogni 2 secondi, caratteristica base del TX20. Quello che deve sorprendere è che nessuna UWP C# è in esecuzione, quindi nessuna riga di codice imposta il DTR GPIO17 a 0 logico, stato necessario per avviare la trasmissione dei datagrammi del TX20 ogni 2 secondi. Prima di analizzare questo comportamento, conviene impostare un tempo di divisione molto basso pari a 5msec e analizzare un altro fenomeno molto importante. La Fig.178 evidenzia che tra l'invio di un datagramma e l'altro, il DSO rileva sempre un'alimentazione pari a circa +3.3V, ma leggermente disturbata da un rumore che spesso si sovrappone al datagramma stesso, anche se questo non si vede dalla figura. La motivazione di questo fattore, alquanto importante, è derivato dall'impiego di un'alimentatore di scarse prestazioni, pagato 2€ sul mercato asiatico, ma che dimostra tutti i suoi limiti. Una tecnica grossolana che permette di ridurre in modo marcato questo disturbo è l'uso di un anello ferromagnetico, come si vede in Fig.179. Avvolgendo il cavo di alimentazione a spire intorno all'anello, si realizza una specie di toroidale riuscendo ad eliminare in modo marcato i disturbi.

Inutile sottolineare che l'alimentatore in dotazione da +5V con erogazione di corrente pari a 2A soddisfa solo sulla carta i criteri di immunità e compatibilità.

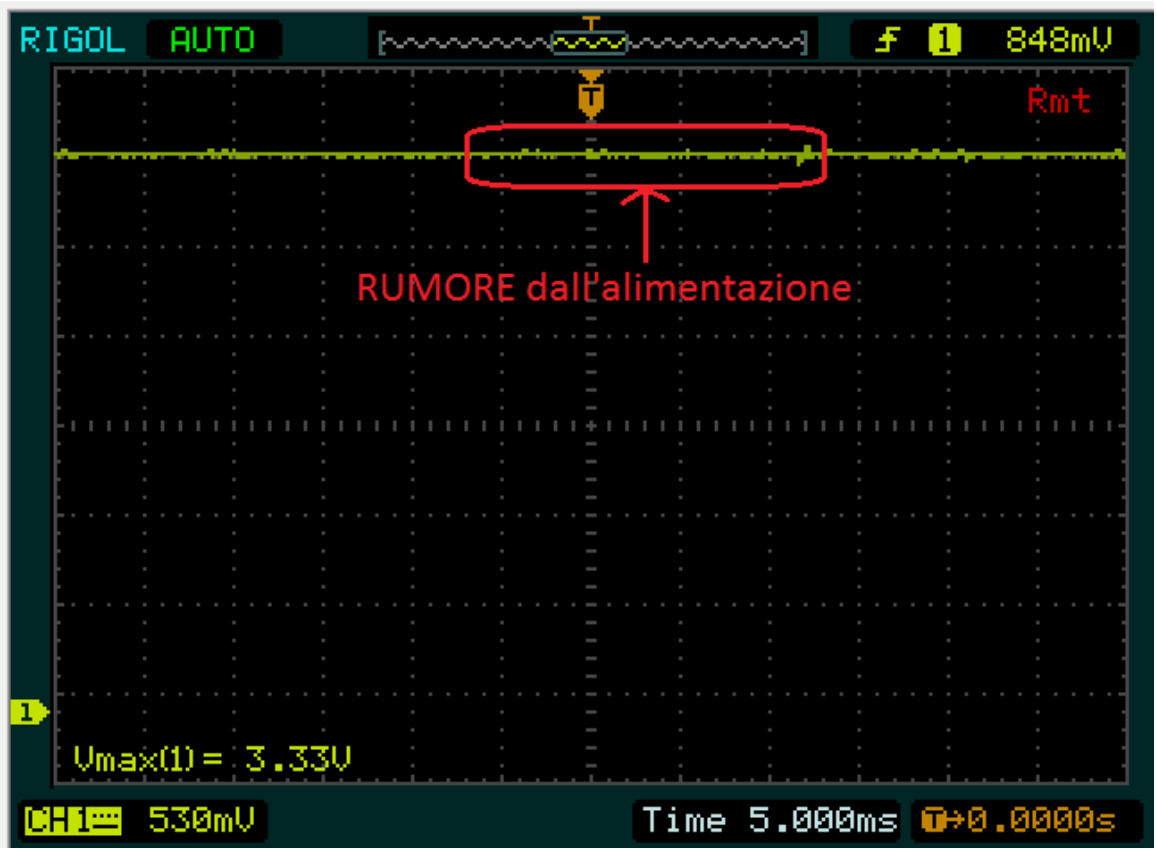


Fig.178

Un dispositivo generico è compatibile se non emette una quantità di disturbo atta ad interferire con altri dispositivi, mentre l'immunità garantisce il corretto funzionamento del device anche in presenza di altri apparecchi che emettono disturbo.

Fig.179

Per il resto dello sviluppo del progetto viene adottato un'alimentatore Aukru sempre da +5V con massima erogazione di corrente pari a 3A. Il costo del device lievita circa verso i 12€. Il risultato di questa sostituzione è visibile chiaramente in Fig.180 nella quale si vede che il segnale di trasmissione è più pulito.

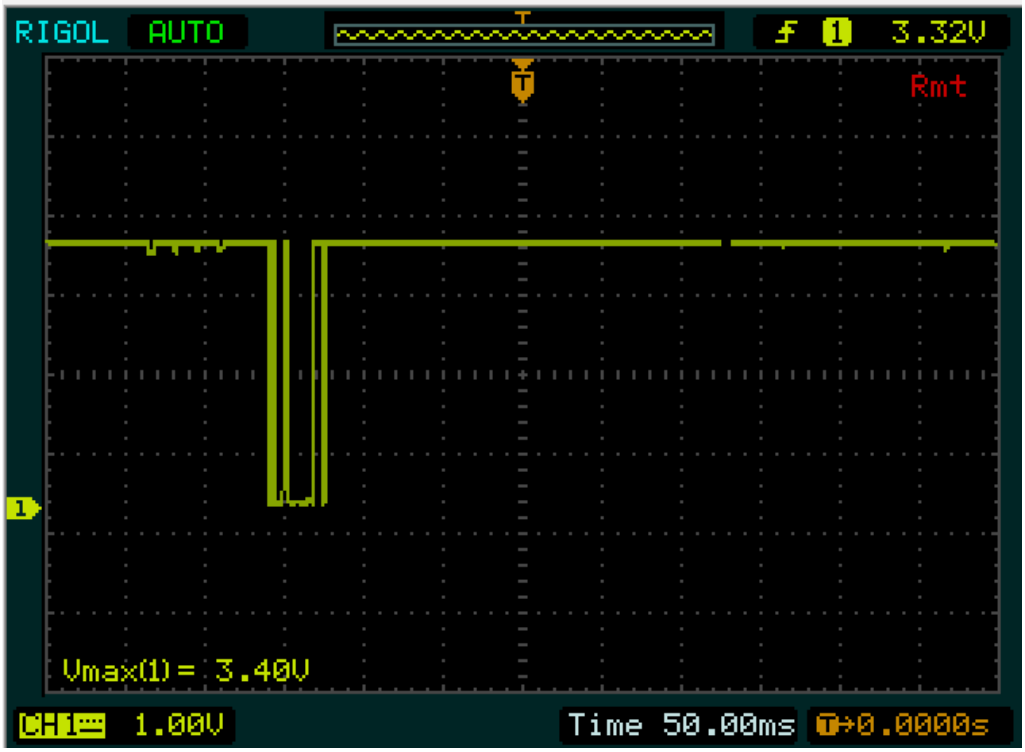


Fig.180

Il problema della trasmissione dell'intero datagramma non è variato ovviamente, e la conferma totale di questo strano comportamento è visibile in Fig.181 con un tempo di divisione pari a 500msec, dal quale è evidente la trasmissione infinita ogni 2 secondi del dati da parte del TX20.

E' possibile analizzare nel dettaglio il datagramma, impostando l'uso di due cursori sul Rigol e abilitando la modalità di memorizzazione del segnale tramite il pulsante "RUN/STOP". In questo modo è possibile vedere la dimensione temporale del datagramma in modo indicativo.

La Fig.182 indica un Δx pari a 39.4msec, ma ovviamente il secondo cursore CurB è stato posizionato in modo indicativo senza potere tenere conto di tutti i bit ad 1 che compongono la coda del datagramma.

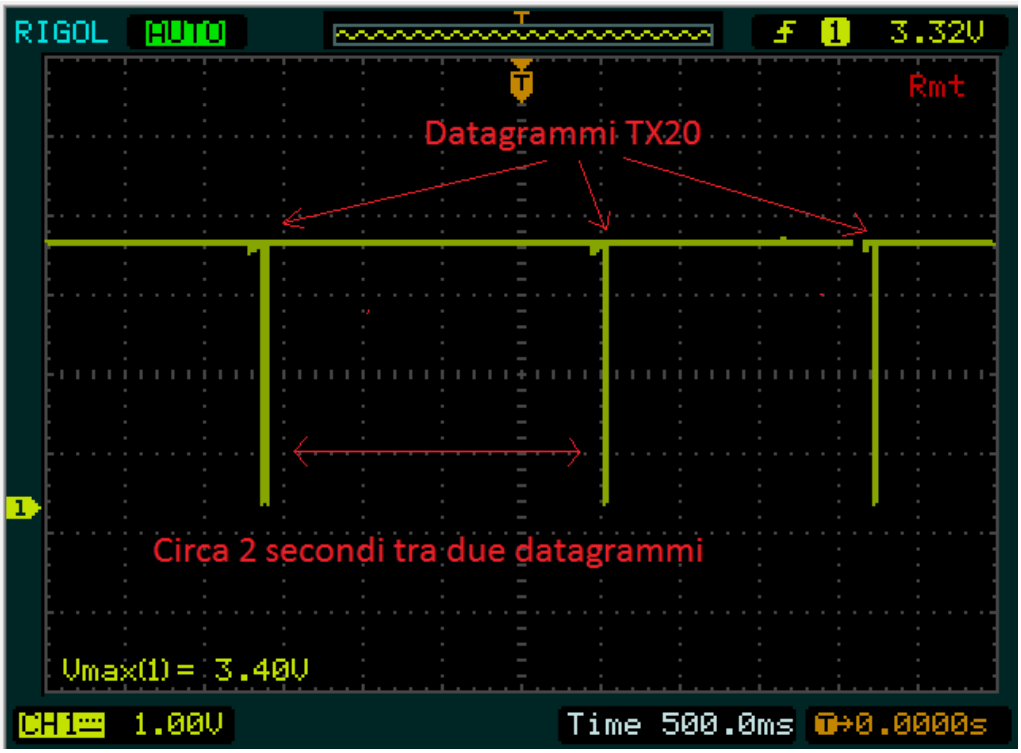


Fig.181

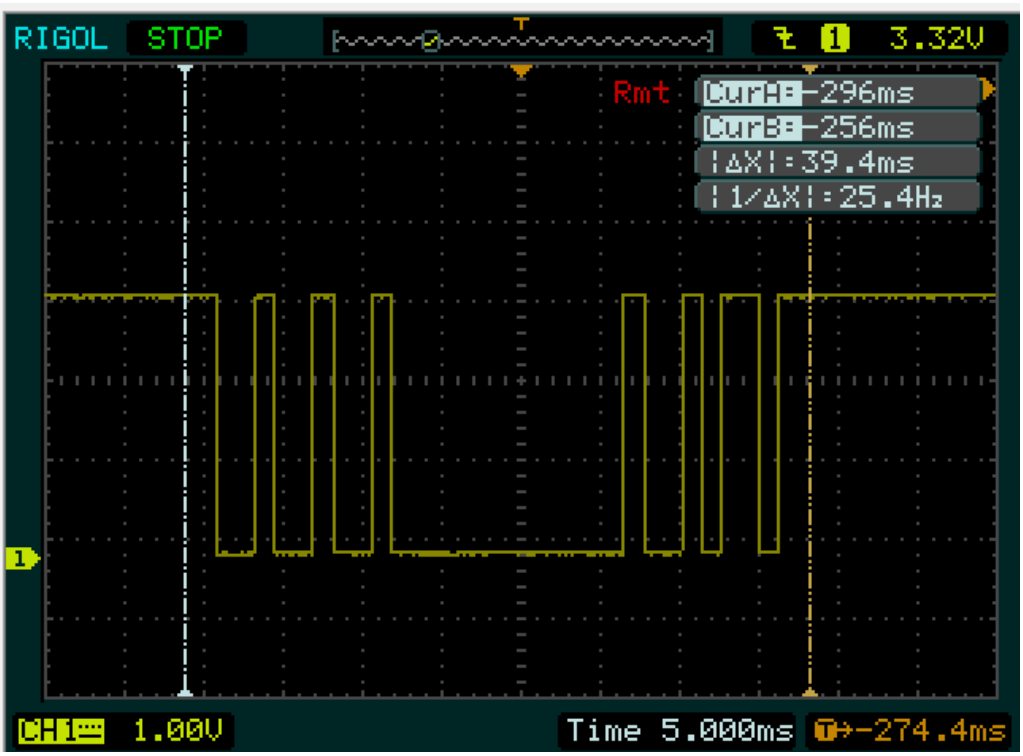


Fig.182

La conclusione di questa breve analisi tramite il DSO, è che il TX20 trasmette perché la linea DTR dell'anemometro rileva lo stato logico basso sul pin GPIO17 del Pi 3 anche in assenza di un qualsiasi tipo di software in esecuzione sul Raspberry. L'unico modo per evitare la trasmissione è impostare uno stato logico di alta impedenza, per poi modificarne lo stato a livello software. Il diagramma a blocchi di Fig.183 riassume la strategia da seguire per avere una corretta implementazione degli stati logici. Nello schema a) si considera il GPIO17 a 0 logico anche in assenza di codice, come evidenziato dal DSO, quindi una apposita circuiteria deve fornire al DTR del TX20 lo stato logico di alta impedenza. In modo duale, nello schema b), quando si decide di programmare via software il GPIO17 a stato logico alto, la medesima circuiteria del blocco a), deve impostare sul DTR lo stato logico basso così da permettere l'invio automatico dei dati.
In definitiva si deve programmare in C# il pin GPIO17 a 1 logico per avviare la trasmissione dei dati dal TX20 al Pi 3, mentre per interromperla basterà reimpostare, sempre sul GPIO17, lo stato logico 0 o, eventualmente, terminare l'applicazione.

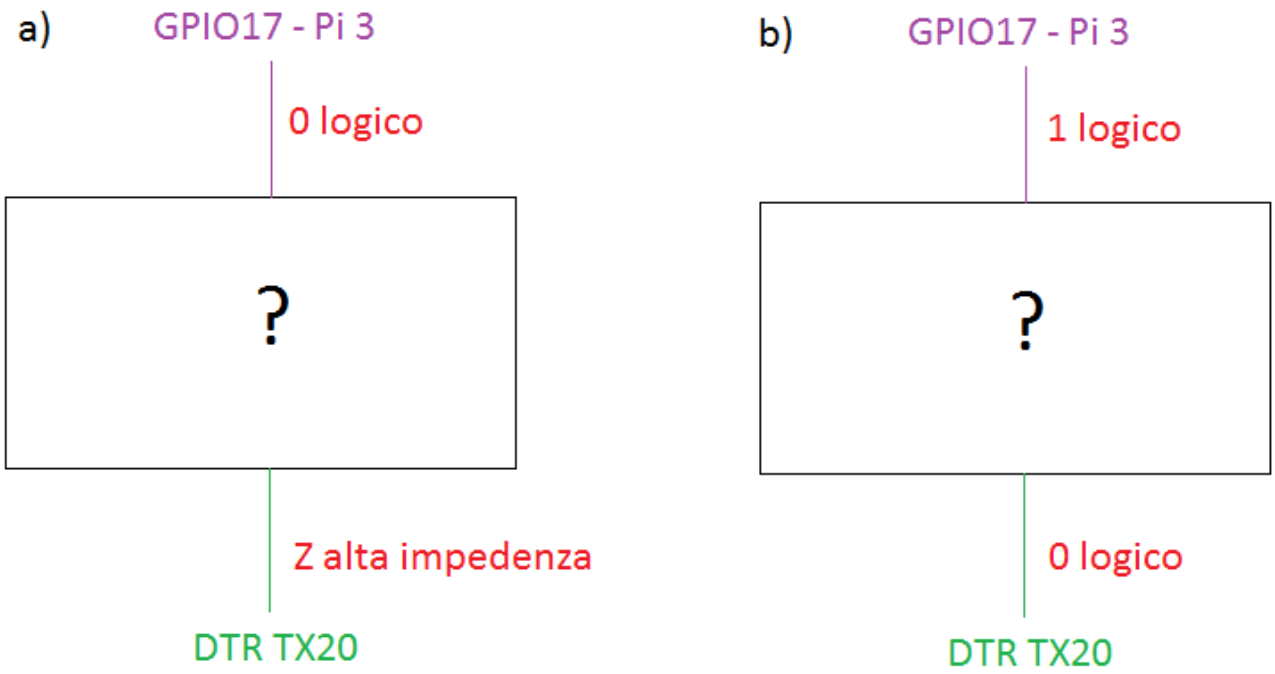


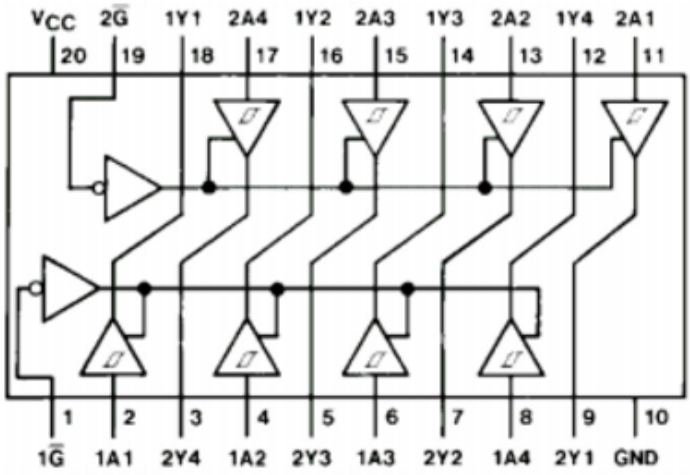
Fig.183

Il blocco circuitale che permettere di risolvere l'adattamento degli stati logici è un buffer/driver a trigger di Schmitt con uscite 3-state.

4.1.4 74xx244

L'integrato 74244 è l'integrato più utilizzato come buffer/driver a trigger di Schmitt con uscite 3-state ed è commercializzato nelle varie famiglie logiche S, LS, AS, ALS, HC e HCT. La Fig.184 mostra la pinatura del chip a 20 pin del 74244 e la tabella di verità, quest'ultima è fondamentale per una corretta implementazione della parte hardware.

Connection Diagram



Function Table

Inputs		Output
$\overline{G}$	A	Y
L	L	L
L	H	H
H	X	Z

L = LOW Logic Level
H = HIGH Logic Level
X = Either LOW or HIGH Logic Level
Z = High Impedance

Fig.184

L'alimentazione Vcc è presente sul pin 20, mentre la massa sul ping 10. Quando l'ingresso attivo basso 1G sul pin numero 1 è, ad esempio, collegato a massa, ossia 0 logico, lo statico logico presente sull'ingresso 1A1 del pin numero 2 viene portato sull'uscita 1Y1 del pin numero 18. Nel momento in cui l'ingresso 1G viene posto a livello logico alto, non ha importanza dello stato logico presente su 1A1, visto che l'uscita 1Y1 sarà sempre in alta impedenza. Il comportamento logico del 74244 è duale rispetto alle esigenze di progetto della Fig.183, la tabella Tab.12 riassume le differenze.

74xx244			74xx244 necessario al progetto		
Input		Output	Input		Output
$\overline{G}$	A	Y	$\overline{G}$	A	Y
L	L	L	L	L	Z
L	H	H	H	L	L
H	X	Z			

Tab.12

L'ingresso 1A1 deve venire collegato direttamente a massa, visto che l'uscita 1Y1 deve valere stato logico basso oppure alta impedenza. Il problema è ottenere alta impedenza nel caso che l'ingresso 1G sia a 0 logico e viceversa. La soluzione è molto semplice, basta invertire lo stato logico del pin GPIO17 tramite un inverter, così da ottenere il comportamento voluto. La Fig.185 mostra in sintesi il circuito completo con le sole porte utilizzate.

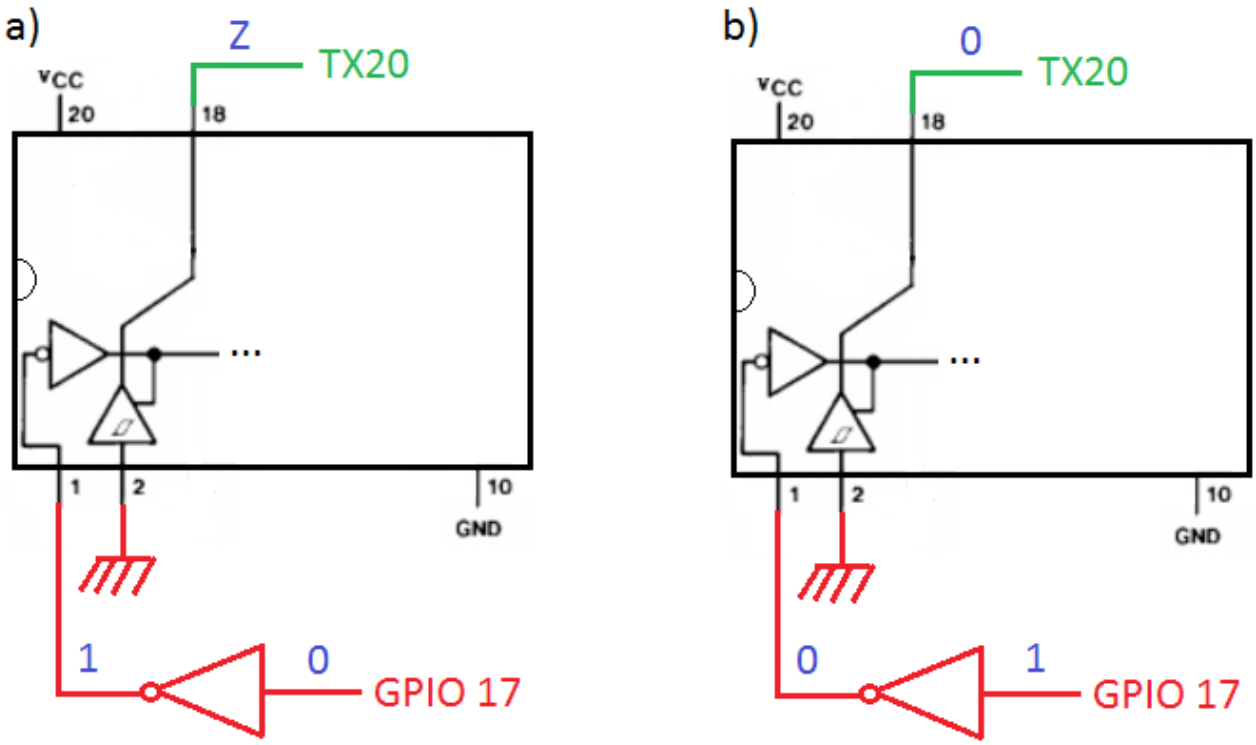
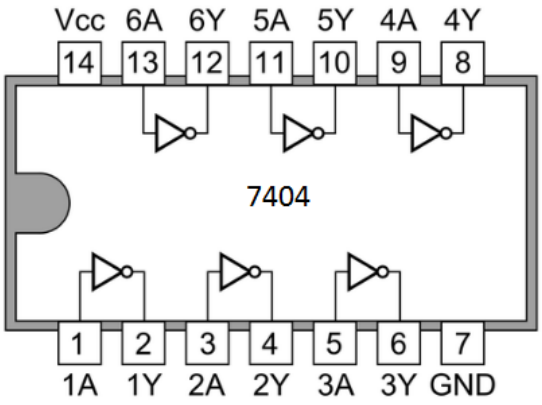


Fig.185

Nel momento in cui la GPIO17 ha uno stato logico che il TX20 vede come attivo basso, ossia 0 logico, l'inverter porta un segnale attivo alto sulla porta numero 1 del 74244, segnale che però verrà invertito e che permette di avere sull'uscita 1Y1 pin 18 lo stato di alta impedenza indipendentemente dal valore logico presente sull'ingresso 1A1 del pin 2. Quando invece la GPIO17 viene programmata con stato logico alto, l'inverter genera lo stato logico basso che verrà poi invertito dal secondo inverter sulla porta 1G, il quale questa volta attiva il trigger di Schmitt facendo in modo che lo stato logico presente sul pin numero 2, ossia 0 logico visto il collegamento diretto a massa, venga portato in uscita sul in 18, in questo modo il DTR del TX20 rileva lo 0 logico ed inizia a trasmettere. Per completezza la Fig.186 riporta la pinatura di un inverter TTL 7404.



Input	Output
A	Y
0	1
1	0

Fig.186

In definitiva il collegamento tra i vari device risulta molto semplice e schematizzabile in Fig.187.

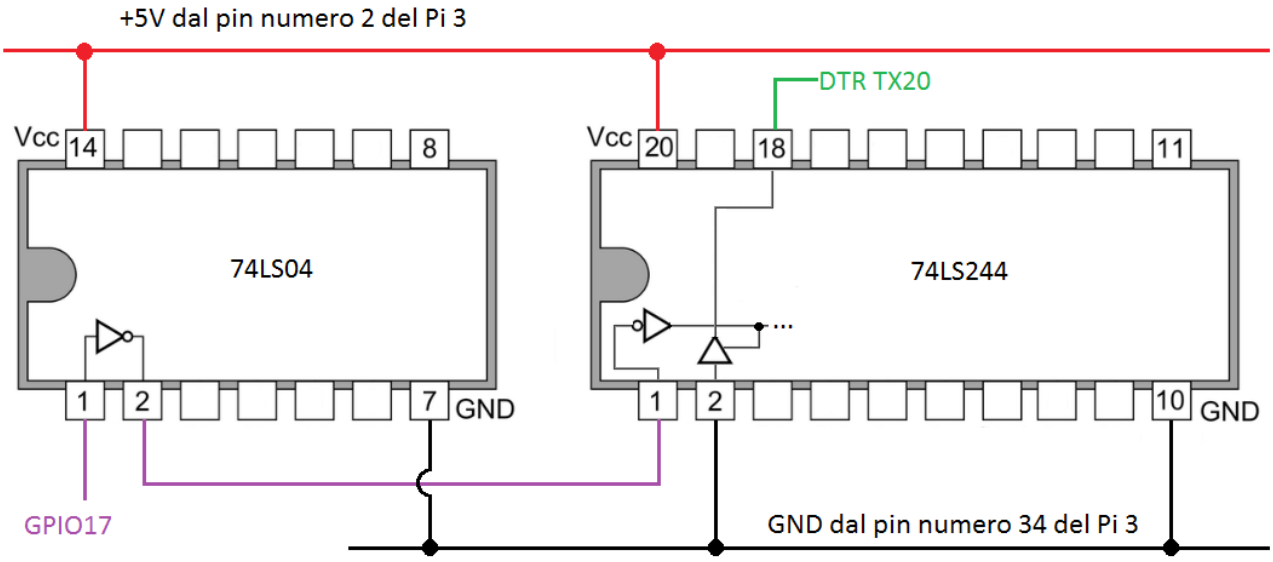


Fig.187

La famiglia logica dei due integrati è la TTL, infatti viene fornita un'alimentazione di +5V prelevata dalla GPIO2 del Pi 3. La Fig.188 mostra la semplice realizzazione del circuito su breadboard.

Il risultato finale all'accensione del Raspberry è la totale assenza di datagrammi trasmessi dal TX20, fino al momento in cui non viene avviata l'apposita UWP C# ancora non implementata. Sarà effettivamente così?

Il DSO, tramite la Fig.189, risponde a questo semplice quesito.

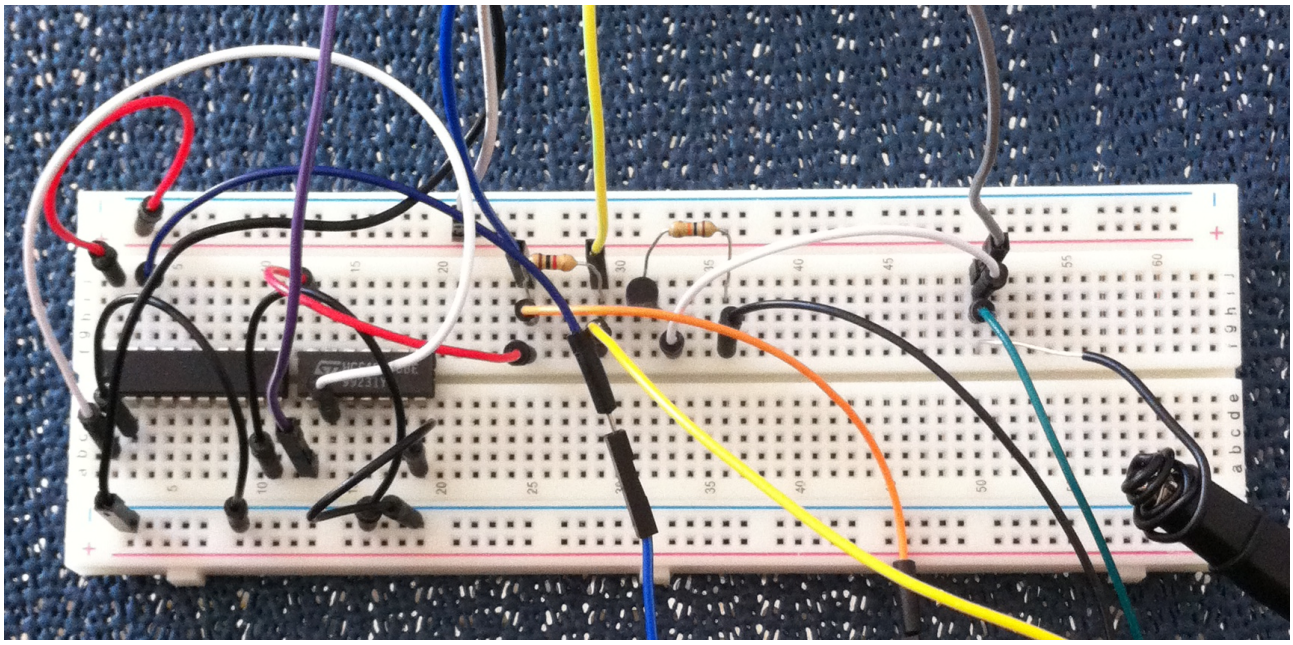


Fig.188

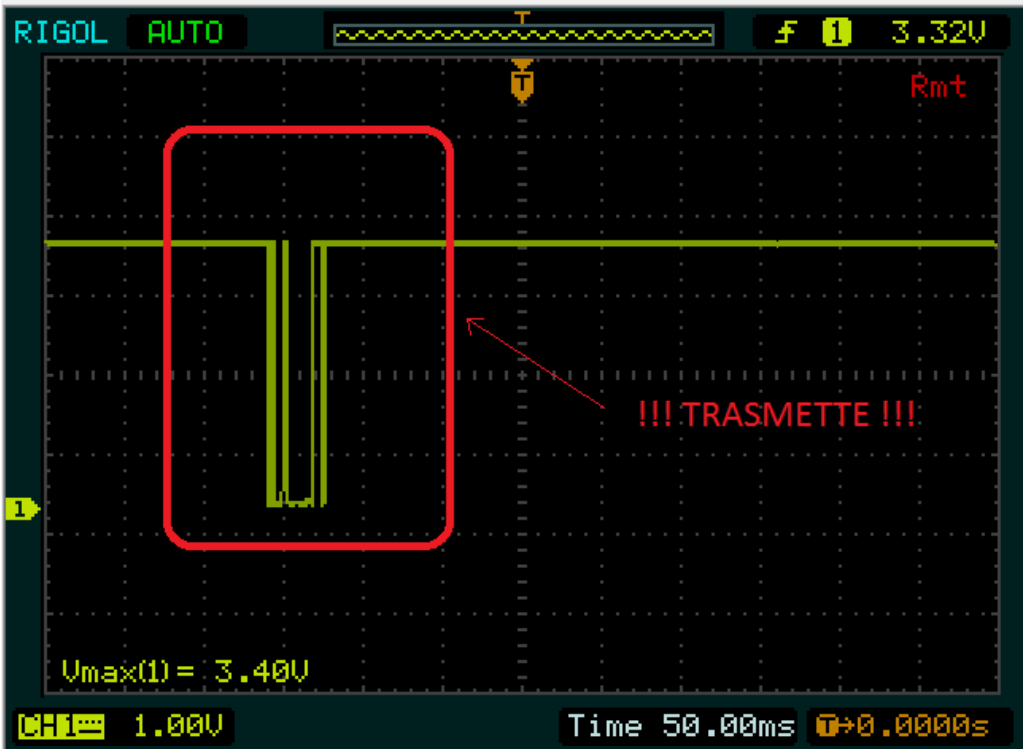


Fig.189

L'anemometro continua a trasmettere nonostante l'inserimento della circuiteria logica per la gestione degli stati. Perché?

Questo tipo di comportamento è legato allo stato di default del pin GPIO e al tipo di famiglia logica utilizzata.

4.1.5 Broadcom BCM2837 ARM

Il Broadcom BCM2837 è un SoC (System-on-Chip) utilizzato nel Raspberry Pi 3 e, come i suoi predecessori BCM2835 e BCM2836, contiene una serie di periferiche come:

- GPIO
- Interrupt controller
- Timers
- USB
- PCM/I2S
- DMA Controller
- I2C Master
- I2C/SPI slave
- SPI0, SP1, SPI2
- UART0
- PWM

Il punto interessante da analizzare è ovviamente la GPIO e sulla base del datasheet ufficiale del 2835, il diagramma a blocchi della GPIO risulta essere quello di Fig.190, anche se quello del 2837 potrebbe avere leggere differenze, in ogni caso non è disponibile online. E' interessante vedere che le porte GPIO sono indistintamente collegate ad un blocco logico che ne rileva lo stato e la variazione dal fronte alto a basso o viceversa, caratteristica quest'ultima che verrà sfruttata a livello di programmazione UWP per capire quando inizia la trasmissione del datagramma inviato dal TX20. Il datasheet del 2835 riporta in modo chiaro gli stati iniziale di tutte le 54 GPIO del SoC, ovviamente il Pi 3 ne sfrutta solo 24, quindi l'analisi è limitata ai soli pin utilizzati dal Raspberry.
La Tab.13 riporta lo stato di default della porta e la funzione di base implementata sul Pi 3, la quale a livello di SoC potrebbe venire modificata tramite codice Assembly ARM. Risulta subito chiaro che la GPIO17 ha uno stato di default "Low", quindi 0 logico, inoltre la porta è configurata in ingresso, caratteristiche che spingono il DTR del TX20, collegato direttamente a tale pin, a rilevare 0 logico trasmettendo subito i datagrammi. E' importante sottolineare che il tipo di elettronica all'interno dell'anemometro non è nota e, la rilevazione dello stato basso, con conseguente invio dei dati, è una ovvia conseguenza da parte del TX20, comportamento che è necessario gestire.

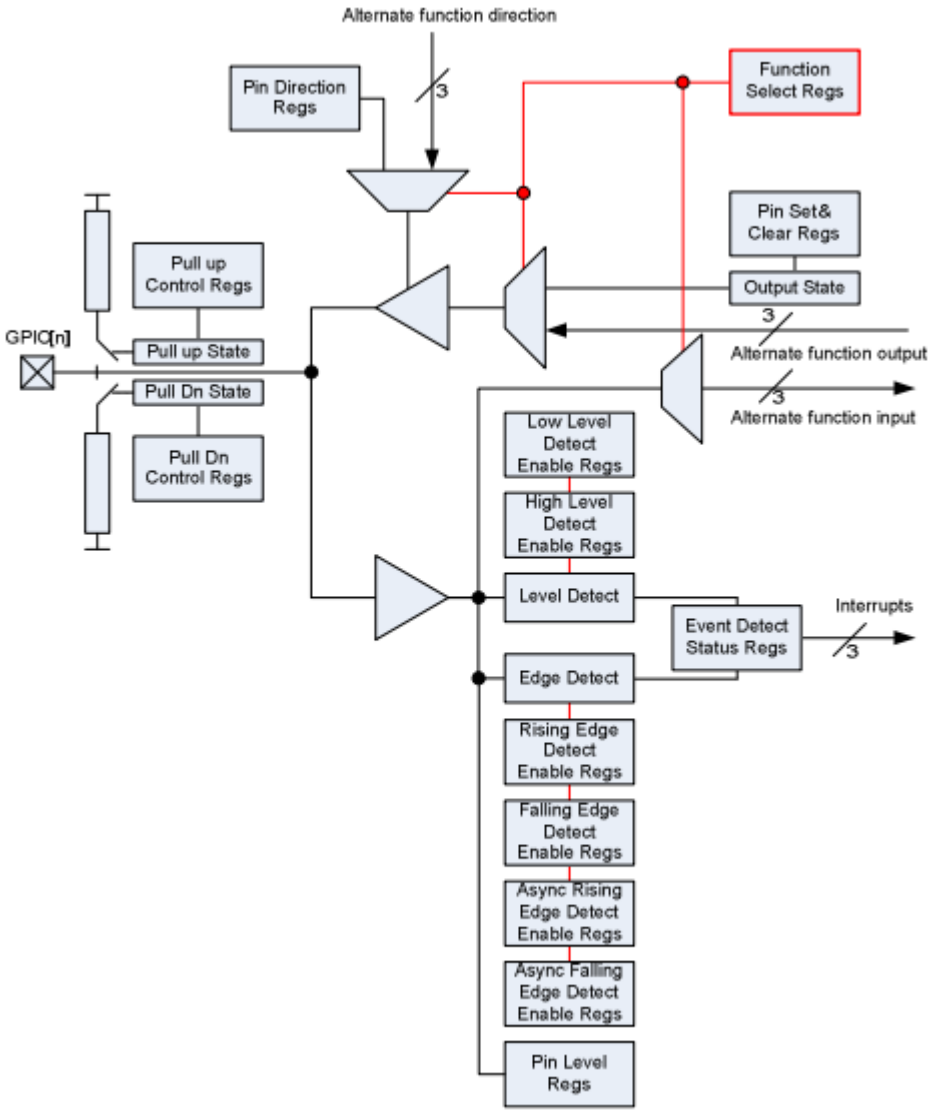


Fig.190

GPIO	*Pull*	*Funzione*
2	High	I2C1 SDA
3	High	I2C1 SCL
4	High	
5	High	
6	High	
7	High	SPI0 CS1
8	High	SPI0 CS0
9	Low	SPI0 MISO
10	Low	SPI0 MOSI

11	Low	SPI0 SCLK
12	Low	
13	Low	
16	Low	SPI0 CS0
17	Low	
18	Low	
19	Low	SPI1 MISO
20	Low	SPI1 MOSI
21	Low	SPI1 SCLK
22	Low	
23	Low	
24	Low	
25	Low	
26	Low	
27	Low	

Tab.13

Tutte le porte GPIO sono in ingresso, questo permette quindi di chiudere la maglia elettrica del canale DTR sul GPIO17 con conseguente invio dei dati, come da Fig.191.

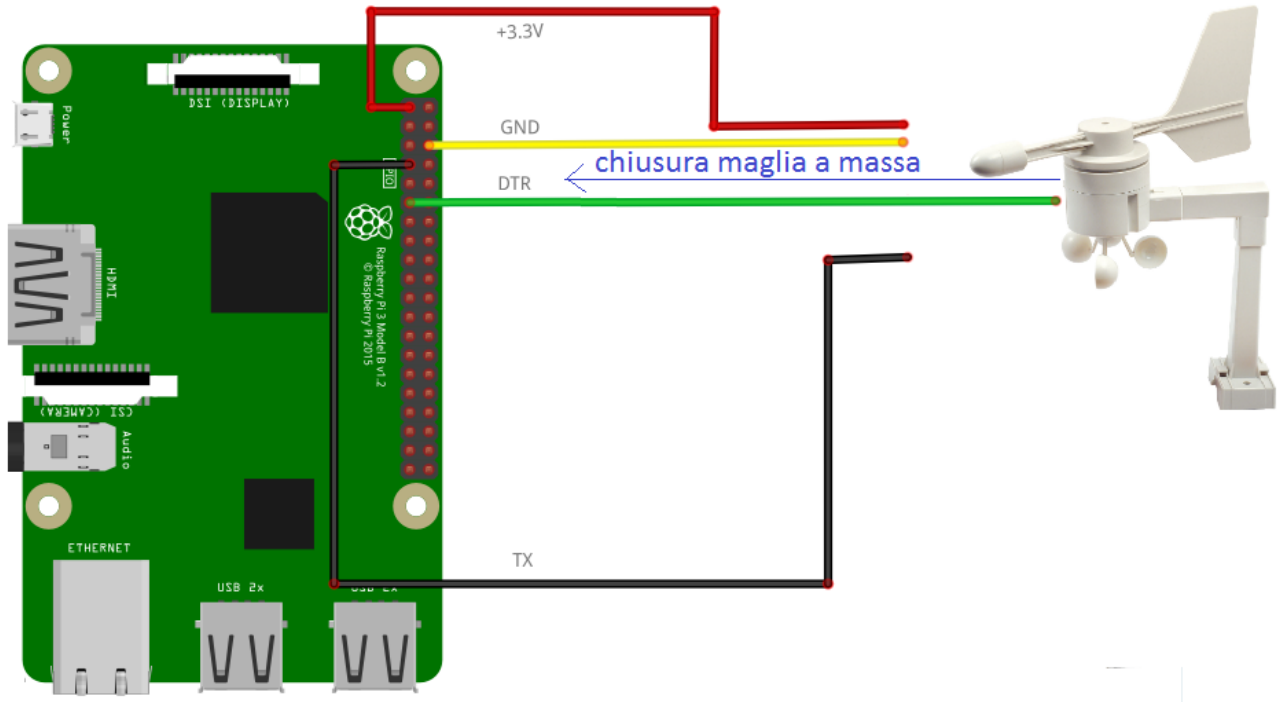


Fig.191

Capita la motivazione dell'invio dati con connessione diretta DTR-GPIO17, resta da capire l'errato comportamento anche in presenza del 74LS04 e del 74LS244 che, sulla carta, avrebbero dovuto risolvere il problema.

Per capire il problema la Fig.192 ricorda la tipologia di connessione tra Pi 3 e gli integrati.

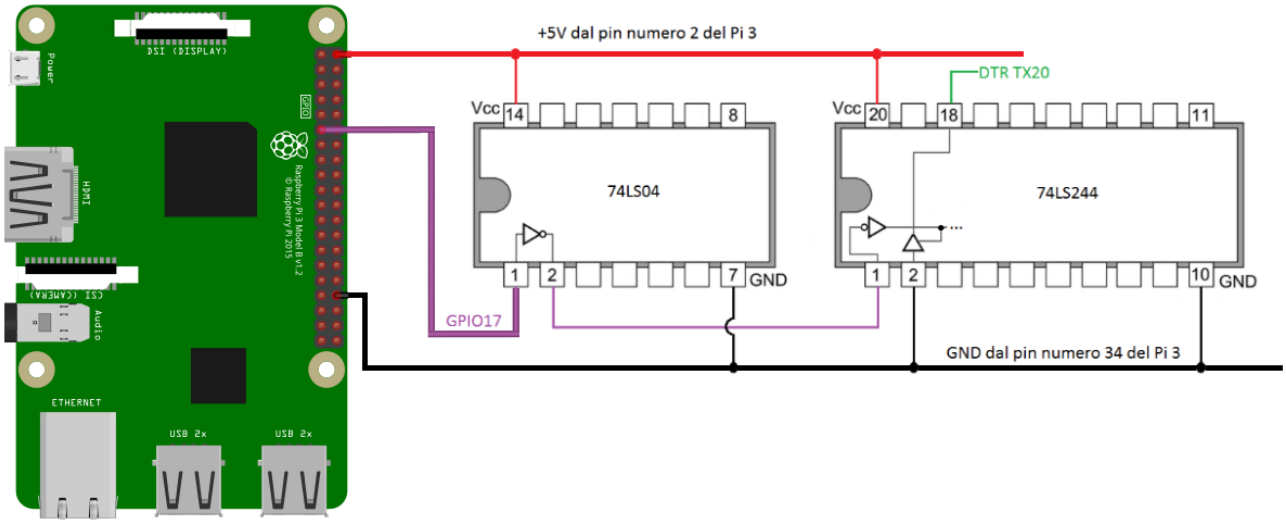


Fig.192

Lo stato di default del pin GPIO17 è 0 logico, quindi l'inverter 74LS04 dovrebbe portare sul pin 2 lo stato logico alto che immesso nel pin 1 del 74LS244 dovrebbe lasciare il canale DTR del TX20 in alta impedenza. Ovviamente questa situazione sulla carta funziona, ma di fatto la precedente Fig.189 dimostra l'incontrario, di conseguenza significa che il DTR sente lo 0 logico ottenibile solamente se il pin 1 del 74LS244 è a stato logico basso, quindi l'ingresso dell'inverter sul pin 1 del 74LS04 è allo stato logico alto. La Fig.193 riassume quanto appena esposto in modo più conciso e chiaro.

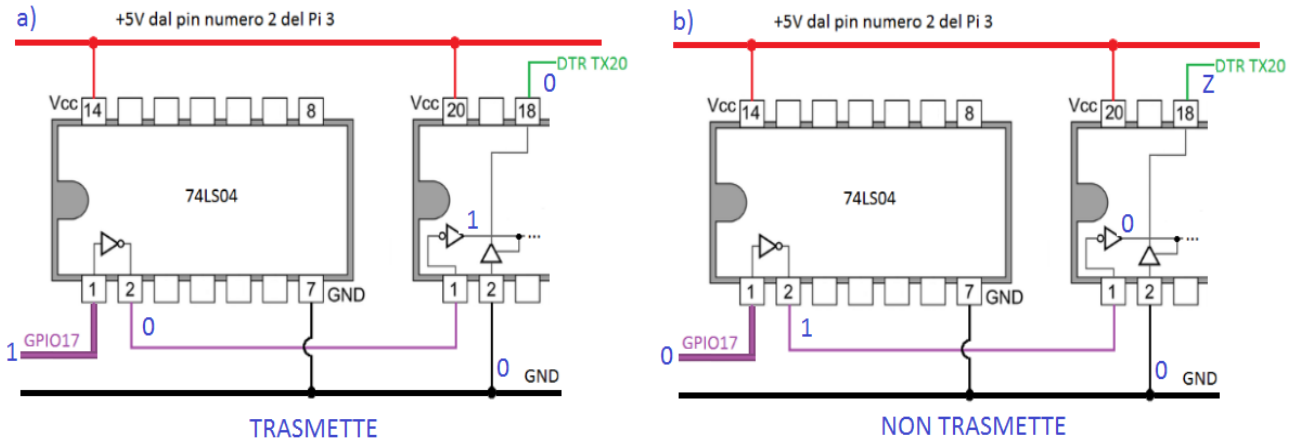


Fig.193

Il pin numero 1 del 74LS04 sento un livello logico alto nonostante la GPIO17 sia per default a livello logico 0. La risposta a questa situazione sta nel fatto che la GPIO17 è si per default a stato logico basso, ma è impostata in ingresso, di conseguenza l'inverter 74LS04, essendo appartenente alla famiglia TTL, sente alta impedenza, ossia cavo staccato, ed in tale circostanza il TTL imposta l'ingresso a livello logico alto. Questo comportamento è una caratteristica di tutti i TTL, quindi l'impiego di una simile tecnologia per la connessione diretta alla GPIO17 è una scelta infelice che non permette di risolvere il problema della trasmissione automatica del datagramma all'avvio elettrico del Pi 3. La più semplice delle soluzioni è modificare la tecnologia dell'inverter, passando da un TTL 74LS04, ad un CMOS HCF4069, il quale è realizzato in modo completamente diverso e garantisce il rilevamento corretto della tensione in ingresso sul pin numero 1. Un inverter 4069 qualora senta in ingresso alta impedenza, considera l'ingresso sul pin 1 come a stato logico basso, mentre se avverte una tensione di +3.3V considera l'ingresso a stato logico alto. Questo comportamento risolve la problematica di invio automatico dei dati, anche se è importante sottolineare che l'integrato HCF4069 ha bisogno di una tensione di alimentazione di +3.3V compatibile con il massimo livello di tensione in ingresso, mentre il 74LS244 necessità per forza di una tensione di alimentazione di +5V e, a fare bene, pure l'ingresso sul pin 1 del 74LS244 dovrebbe essere attorno ai +5V per garantire il corretto rilevamento dello stato logico alto, tensione però che non è raggiungibile in uscita dal 4069 proprio perché è alimentato al massimo a +3.3V. In ogni caso un TTL rileva lo stato logico alto sopra i +2V, quindi l'interfacciamento HCF4069-TTL74244 funziona. Quando si implementano circuiti digitali con integrati di famiglie logiche diverse è necessario avere la massima attenzione perché la compatibilità tra le varie famiglie non è garantita, quindi si ha il rischio concreto di avere sulla carta un progetto che logicamente funziona, ma che una volta implementato presenta notevoli incompatibilità elettriche con "anomalie" imprevedibili o, peggio ancora, con un funzionamento errato rispetto alle specifiche richieste. Il paragrafo che segue affronta brevemente quanto affermato in quest'ultime righe proprio per convincere il lettore che non basta avere la porta logica corretta, ma anche la o le famiglie logiche adeguate ai livelli di tensioni richieste e alla gestione dell'alta impedenza.

4.1.6 Le famiglie logiche

Come anticipato nel paragrafo precedente, non basta inserire in un progetto di elettronica digitale la porta logica che soddisfa i requisiti, ma è altrettanto importante considerare le famiglie logiche con cui lavorare. Questo stesso progetto ha evidenziato la necessità di utilizzare una prima porta CMOS HCF ed una seconda porta BJT TTL, quindi non bisogna mai avere la certezza che in un progetto si userà una ed un solo tipo di tecnologia. Le tue principali tecnologie di transistor sono il BJT (Bipolar Junction Transistor), transistor a giunzione bipolare ed il MOS (Metal-Oxide Semiconductor) transistore unipolare a metallo, ossido e semiconduttore. Le caratteristiche che differenziano questi due transistor sono innumerevoli, ma la più importante riguarda la tipologia di lavoro, che nel BJT è a corrente, mentre nel MOS a tensione. Ovviamente questo argomento potrebbe essere talmente espanso da richiedere un libro di testo ad hoc, ma quello che si vuole evidenziare è l'uso di queste due tecnologie a transistor per la realizzazione delle porte logiche e la loro compatibilità elettrica, punto cardine in fase di implementazione e realizzazione di un progetto digitale. Sulla base della scelta nell'uso di BJT o CMOS (Complementary MOS) le famiglie logiche sono classificate come da elenco puntato.

CMOS

- HC/HCT/HCF
- serie 4000

BJT

- TTL (Transistor Transistor Logic)
  - STD (Standard)
  - LS (Low Power Schottky)
  - S (Schottky)
  - ALS (Advanced LS)
  - AS (Advanced S)
  - F (Fast)
- ECL (Emitter Coupled Logic)

La scelta della famiglia logica è legata ad importantissimi fattori, come il consumo energetico e la velocità. La Tab.14 riporta indicativamente i dati della potenza dissipata a riposo dalla porta e la frequenza massima utilizzabile per un segnale ad onda quadra.

	4000	HC	STD	LS	S	ALS	AS	F
PD(mW)	0.001	0.025	10	5	20	1	8	4
Fmax(MHz)	10	50	35	45	125	70	200	130

Tab.14

Risulta evidente che i CMOS sono energeticamente molto convenienti, infatti l'inverter 4069 (serie 4000) ha un consumo di tre ordini di grandezza inferiore rispetto al 74LS04. Il vantaggio dei TTL è la loro enorme velocità, come si vede chiaramente nella serie AS. Questi aspetti sono ovviamente da considerare, ma non solo, visto che l'aspetto più importante è la compatibilità elettrica tra le varie famiglie. Prima di elencare alcuni valori, è necessario ricordare quali sono i parametri elettrici in gioco, attori che molto spesso non vengono presi in considerazione se non dopo aver realizzato e montato nel modo errato il circuito digitale. La Tab.15 riporta in sintesi queste informazioni.

Parametro	*Significato*
V_{cc}	Tensione di alimentazione
T_p	Tempo di propagazione
T_f	Tempo di transizione
$V_{iL,max}$	Valore massimo della tensione in ingresso per avere lo stato basso (low)
$V_{iH,min}$	Valore minimo della tensione in ingresso per avere lo stato alto (high)
$V_{oL,max}$	Valore massimo della tensione in uscita per avere lo stato basso (low)
$V_{oH,min}$	Valore minimo della tensione in uscita per avere lo stato alto (high)
$I_{iL,max}$	Valore massimo di corrente in ingresso allo stato basso (low)
$I_{iH,max}$	Valore massimo di corrente in ingresso allo stato alto (high)
$I_{oL,max}$	Valore massimo di corrente in uscita allo stato basso (low)
$I_{oH,max}$	Valore massimo di corrente in uscita allo stato alto (high)

Tab.15

Sfogliando i vari datasheet delle case produttrici sono ricavabili i dati presenti in Tab.16.

	F_{max}	$V_{iL,max}$	$V_{iH,min}$	$V_{oL,max}$	$V_{oH,min}$	$I_{iL,max}$	$I_{iH,max}$	$I_{oL,max}$	$I_{oH,max}$
ser 4000	10MHz	1.5V	3.5V	0.4V	4.6V*	-0.1µA	0.1µA	0.52mA*	-0.52mA*
HC	50MHz	1.35V	3.15V	0.4V	3.7V*	-0.1µA	0.1µA		
HCT	50MHz	0.8V	2V	0.4V	3.7V*	1µA	1µA		
LS	45MHz	0.8V	2V	0.5V	2.7V	-0.36mA**	20µA	8mA	-0.4mA
STD	35MHz	0.8V	2V	0.4V	2.4V	-1.6mA**	40µA	16mA	-0.4mA
(*) con tensione di alimentazione a +5V (**) la corrente è negativa perché esce									

Tab.16

Un altro importante fattore è la tensione di alimentazione, che può variare in modo molto significativo con la tecnologia CMOS come si vede in Tab.17.

Tecnologia	Tensione di alimentazione
TTL	+5V ±5%
CMOS	da +3V a +20V

Tab.17

I dati in tabella devono spingere il progettista a chiedersi in che modo è garantita o meno la compatibilità elettrica tra le varie famiglie, punto centrale di tutto questo paragrafo. Per rispondere definitivamente a tale quesito conviene analizzare la Fig.194 e la Fig.195.

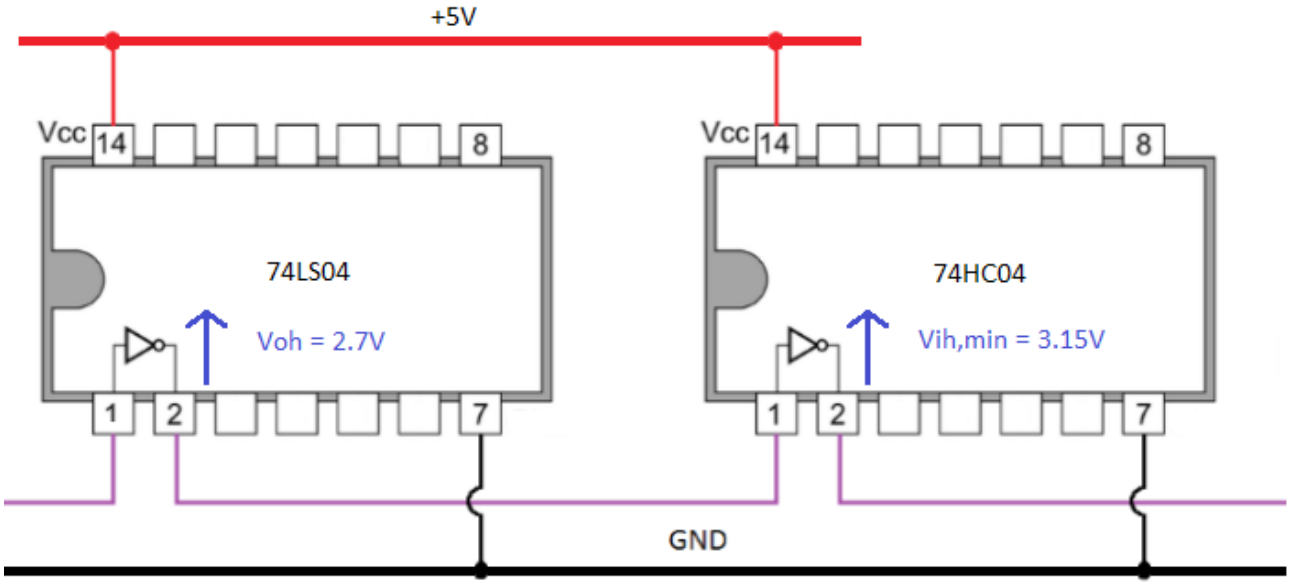


Fig.194

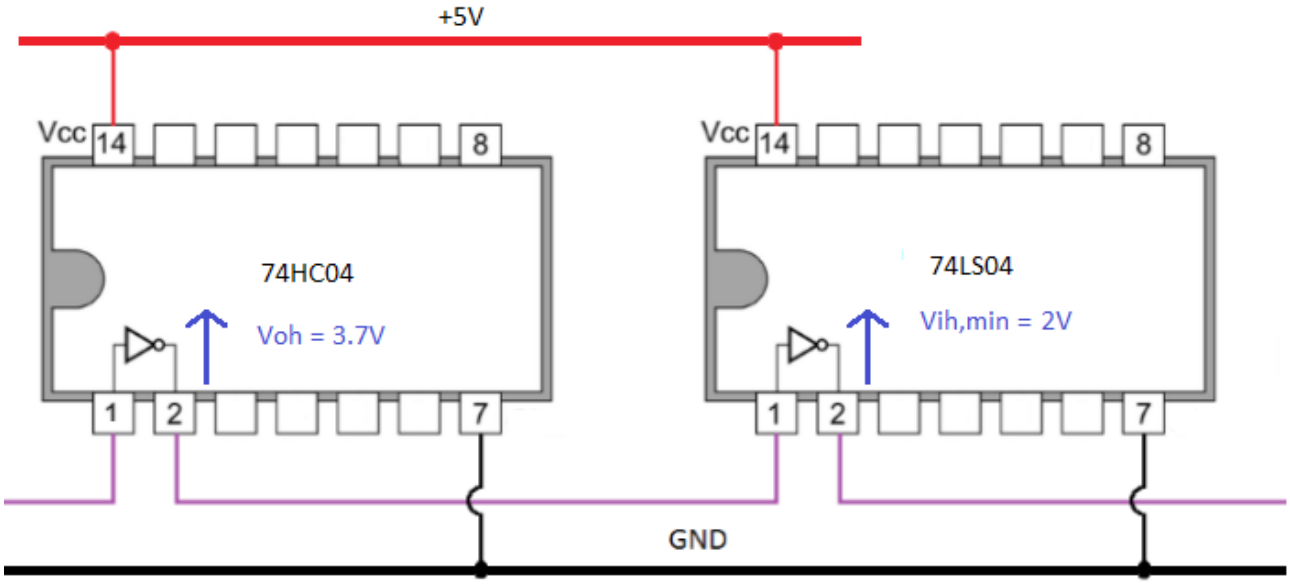


Fig.195

Osservando la Fig.194 si nota che la tensione in uscita V_{oh} pari a +2.7V prodotta dal TTL di tipo LS, non è compatibile con la tensione minima in ingresso $V_{ih,min}$ del CMOS HC che richiede almeno +3.15V per riconoscere lo stato logico alto. Questo aspetto chiude definitivamente il discorso, assicurando che un integrato TTL non può mai pilotare un CMOS.

In Fig.195 l'uscita del CMOS HC produce una Voh pari a +3.7V, tensione compatibile con $V_{ih,min}$ del TTL LS pari a +2V. Un integrato CMOS può sempre pilotare un TTL.

Nel progetto verrà utilizzato, come primo test, un inverter HCF4069 che pilota direttamente il 74LS244, essendo questi gli unici integrati a disposizione dello scrivente. La Fig.196 riporta in sintesi la scelta dei due integrati, facendo molta attenzione a variare la tensione di alimentazione tra i due integrati. La tensione massima prodotto in uscita dalla GPIO17 è pari a +3.3V, caratteristica fondamentale del Pi 3, quindi per avere la certezza che il chip HCF4069 riconosca correttamente lo stato alto, è opportuno fornire una tensione di alimentazione anch'essa pari a +3.3V. Il discorso cambia per l'integrato 74LS244, che essendo un chip appartenente alla famiglia TTL, necessità di un'alimentazione pari a +5V per assicurare il riconoscimento corretto di una tensione che rappresenti il livello logico alto. Qualora si fornisse erroneamente una tensione di alimentazione di +3.3V al chip TTL, il circuito digitale non funzionerebbe in modo corretto. Il Raspberry Pi 3 fornisce direttamente una tensione di +5V sul pi numero 2 della GPIO, come si vede chiaramente in Fig.196.

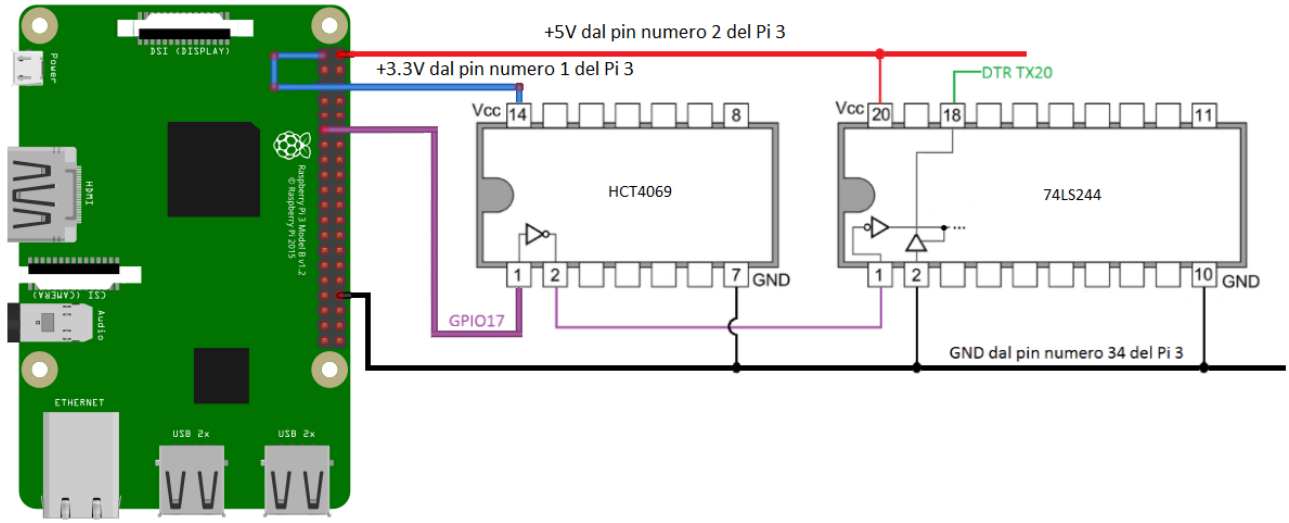


Fig.196

Il montaggio in breadboard è abbastanza semplice come si vede dalla Fig.197.

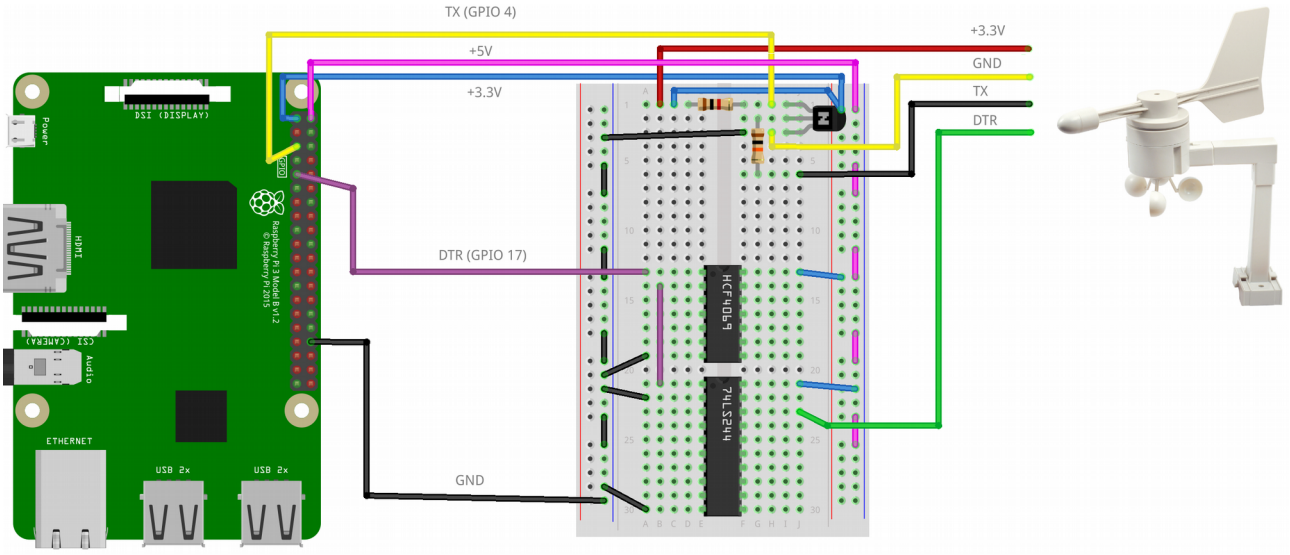


Fig.197

La Fig.198 riporta lo schematics realizzato sempre con il software fritzing. Per comodità di lettura e di debugging si è deciso di utilizzare lo stesso colore dei cavi impiegati nella fase di montaggio.

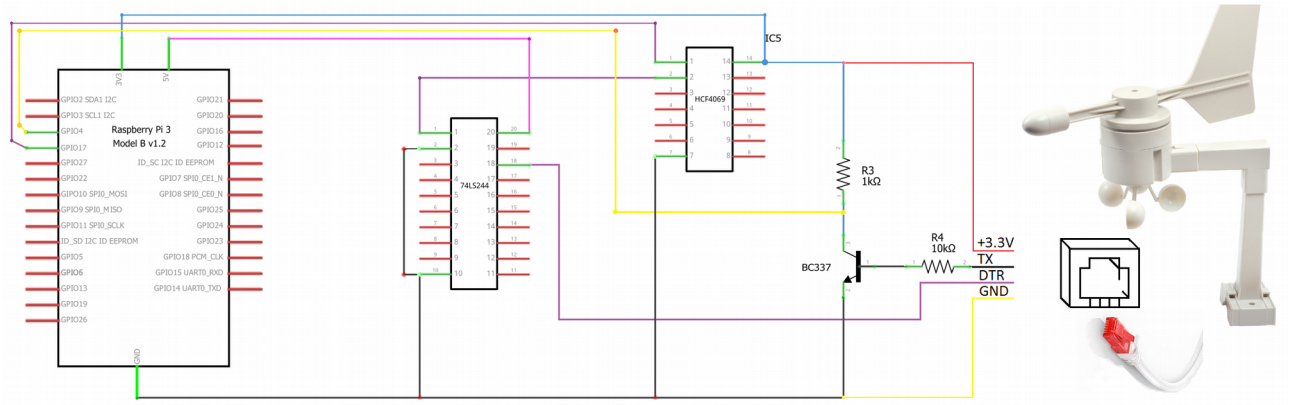


Fig.198

Come già accennato in precedenza, la scelta degli integrati HCF4069 e 74LS244 è stata fatta in base al materiale a disposizione dello scrivente ed anche per la semplice reperibilità di queste parti elettroniche.

Lo step successivo prevede la realizzazione del PCB partendo ovviamente da uno schematics, ma conviene prima togliere la breadboard e vedere in che modo realizzare l'interfacciamento tra il Pi 3 e la circuiteria di gestione del TX20. La Fig.199 riassume il lavoro da svolgere, nel quale si vede l'impiego della GPIO pin 34 come massa al posto del pin numero 6 utilizzato fino ad ora.

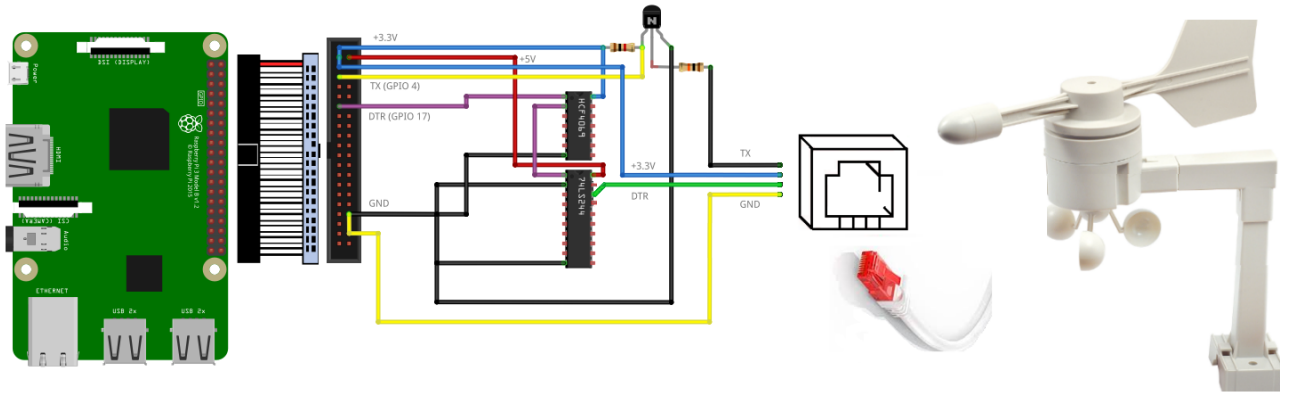


Fig.199

Ovviamente questa modifica non ha nessuna ripercussione sul funzionamento elettrico dell'intero circuito. Il cavo di connessione Pi 3 e scheda PCB è un cavo flat 40 poli utilizzato per anni nei personal computer con la tecnologia PATA. Il connettore Molex 20x2 poli ha la stessa pinatura della GPIO del Pi 3.

Per realizzare il PCB si è deciso di abbandonare il software fritzing a favore del software di AutoDesk Eagle 8.0.2 versione Free. La motivazione di questo passaggio è la presenza di molte librerie già create ad hoc, come ad esempio quella del connettore RJ11 4 poli rilasciata da Ardafruit per Eagle. Indubbiamente esistono molti altri software professionali per la creazione del PCB, come il famosissimo OrCAD 17, rilasciato anche in versione Lite per studenti, la scelta deve comunque considerare la presenza o meno di librerie già pronte per i componenti del circuito. Quasi la totalità dei software PCB non dispongono di librerie per il connettore RJ11 4 poli, quindi l'utilizzatore dovrebbe realizzare il fingerprint PCB di proprio pugno. Le librerie extra inserite in Eagle sono:

- con-rj.lbr per connettori RJ11, RJ12, RJ14, RJ22, RJ25 e RJ45

  https://github.com/zumbik/Eagle-Libraries/blob/master/con-rj.lbr

- pacchetto SparkFun Eagle Libraries Master

  https://github.com/sparkfun/SparkFun-Eagle-Libraries

Il software Eagle utilizzato è la versione 8.0.2 Free per usi non commerciali scaricabile direttamente dal sito ufficiale di Autodesk www.autodesk.com. Una volta installato il software basta registrarsi per richiedere la licenza Free e viene avviato in automatico il Control Panel di gestione di Eagle. Si veda la Fig.200.

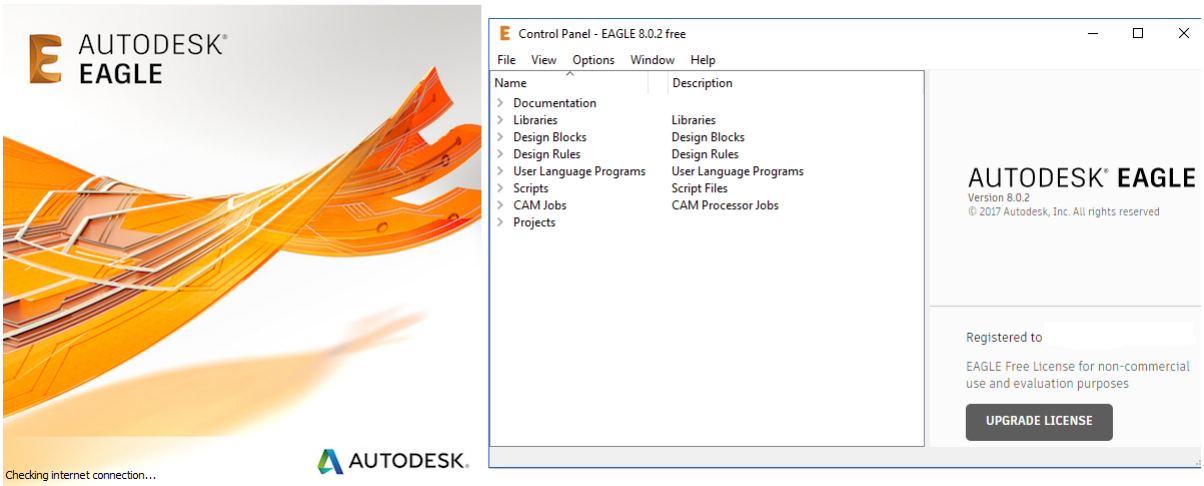


Fig.200

Il primo steps comporta ovviamente la creazione dello schema elettrico, quindi è necessario definire lo schematic aprendo un foglio di lavoro "Schematic" direttamente dall'opzione "New" del menù "File". Una volta aperto il foglio di lavoro si procede subito al salvataggio del file caricando le librerie extra "con-rj.lbr" e tutto il contenuto del pacchetto SparkFun tramite l'icona "Use library" della Fig.201. Successivamente si indica la libreria oppure l'intero contenuto del pacchetto come da Fig.202 e Fig.203.

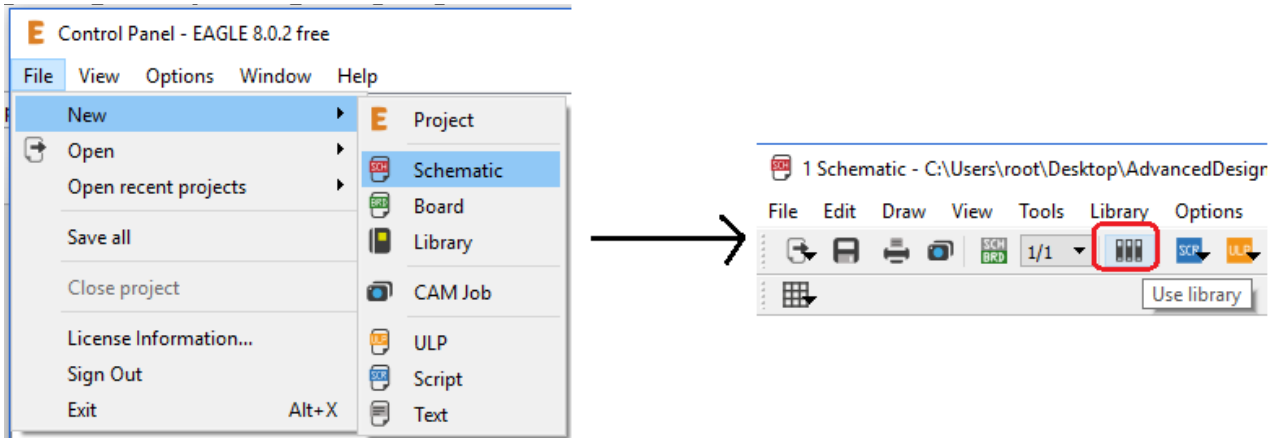


Fig.201

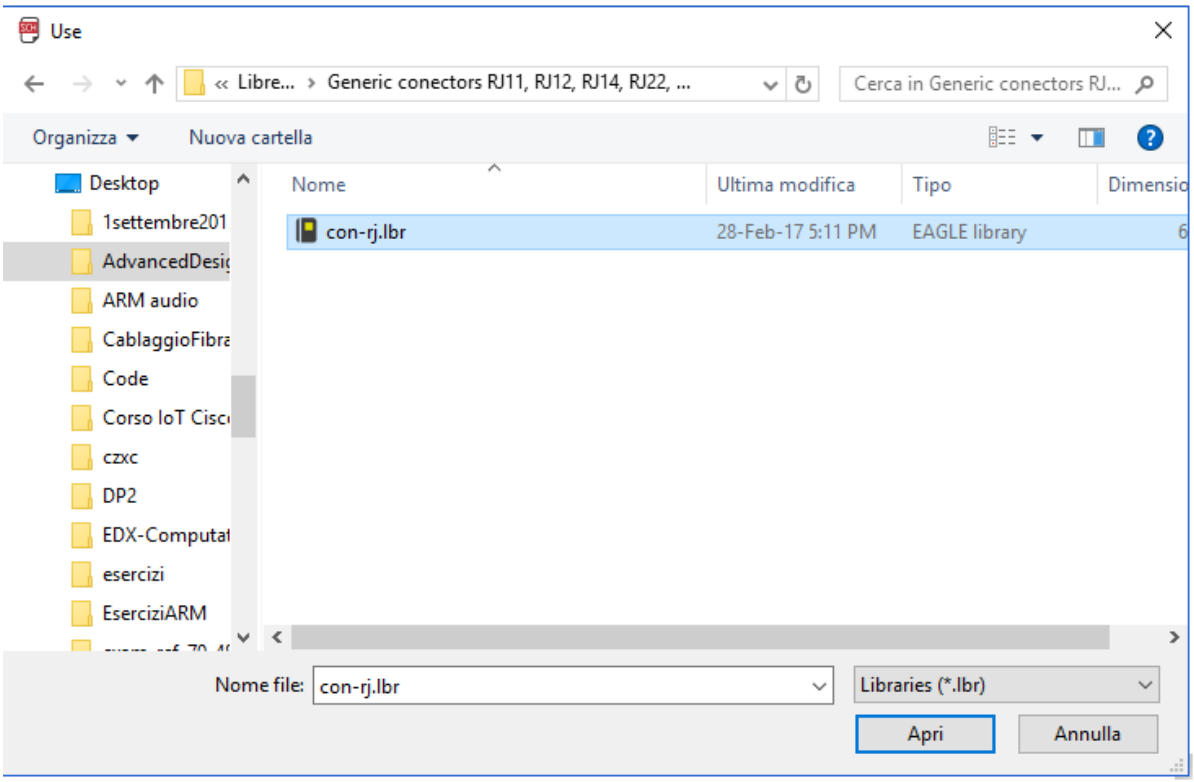


Fig.202

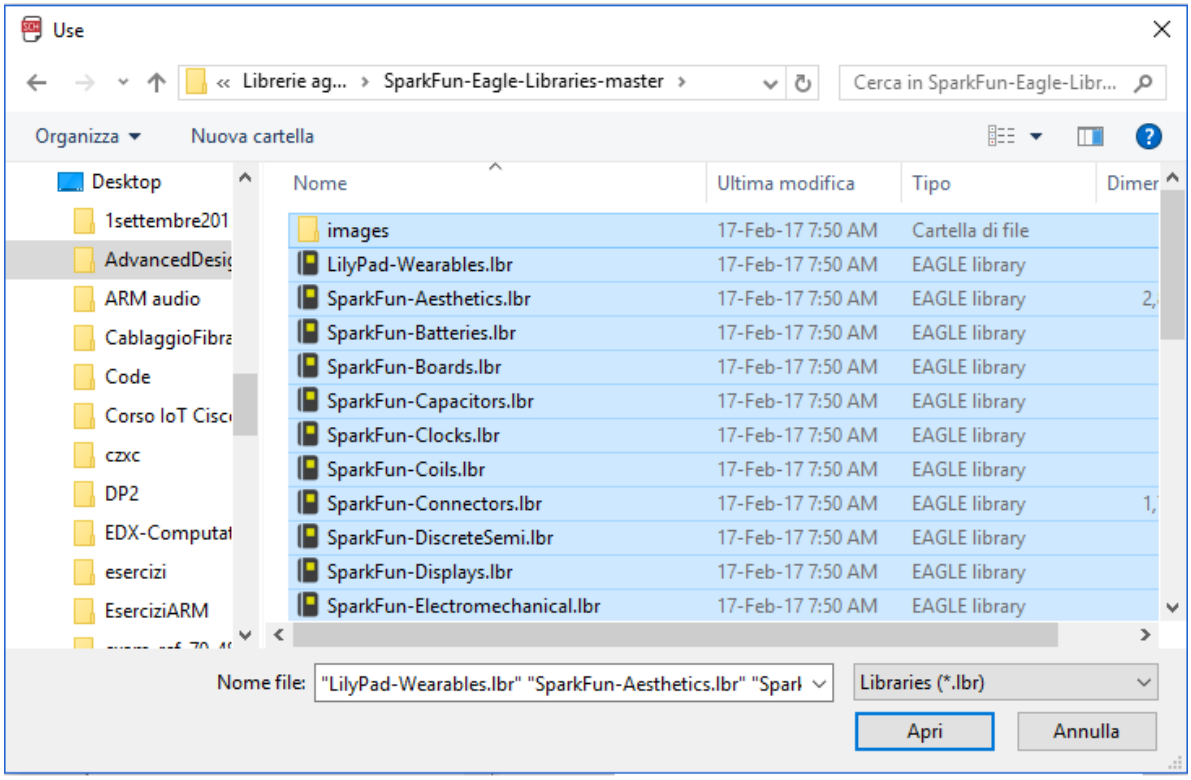


Fig.203

La realizzazione dello schematic è immediata grazie sopratutto alle librerie extra appena caricate che facilitano la creazione del circuito nella posa del connettore Molex 20x2 poli e del connettore RJ11 a 4 poli. La Fig.204 mostra lo schema finale.

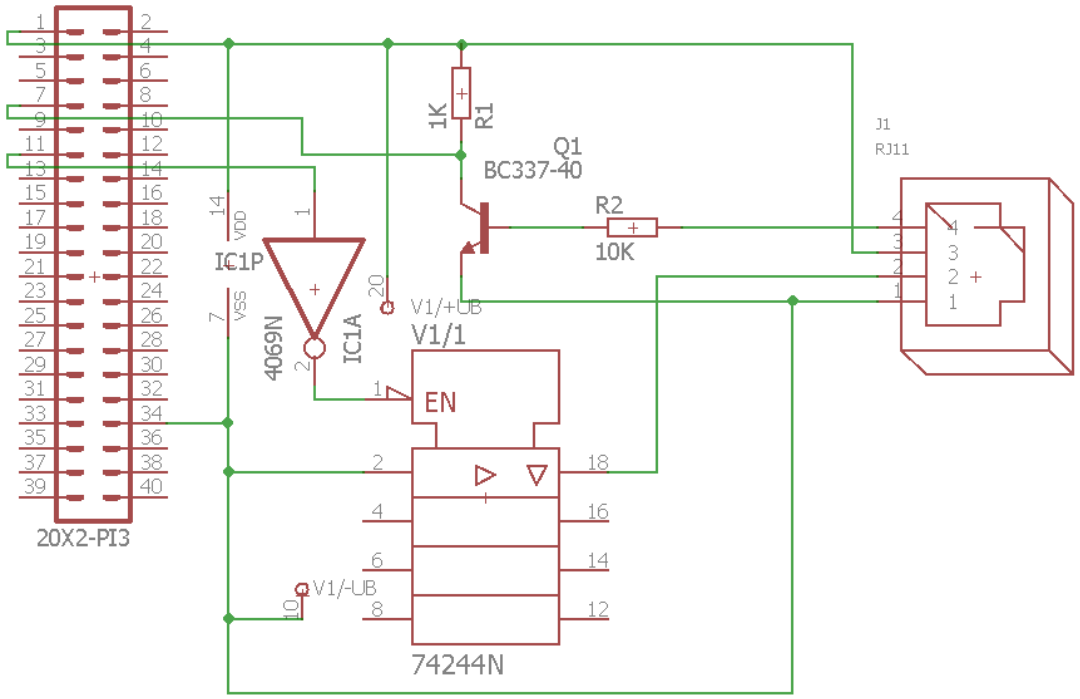


Fig.204

Il passaggio allo schema PCB avviene con l'ausilio dell'icona "Generate/switch to board" come si vede in Fig.205.

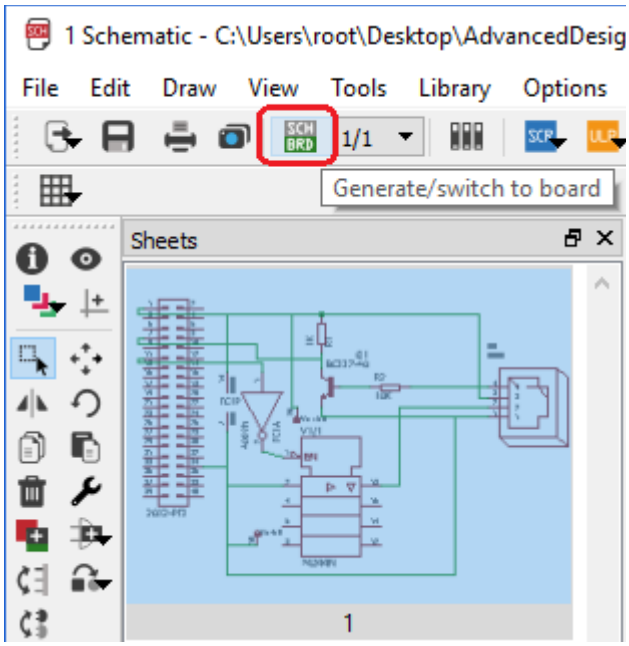


Fig.205

Come fatto per lo schematics si tralasciano tutti i passi di realizzazione della scheda PCB, ma è importante sottolineare che la logica di progettazione deve tenere conto di come poi verrà fisicamente realizzata la scheda. Esistono molti siti online specializzati nella realizzazione di schede stampate, anche se i costi non sono proprio abbordabili. La scheda verrà realizzata in casa, utilizzando una basetta ramata in vetronite di dimensione 10cmX16cm mono faccia sulla quale viene foto incisa, grazie ai raggi ultravioletti, lo schema PCB. Questa operazione viene svolta per circa 200 secondi tramite l'ausilio di un bromografo modello Vacuum Pro Ma. In Fig.206 alcune fasi della lavorazione.

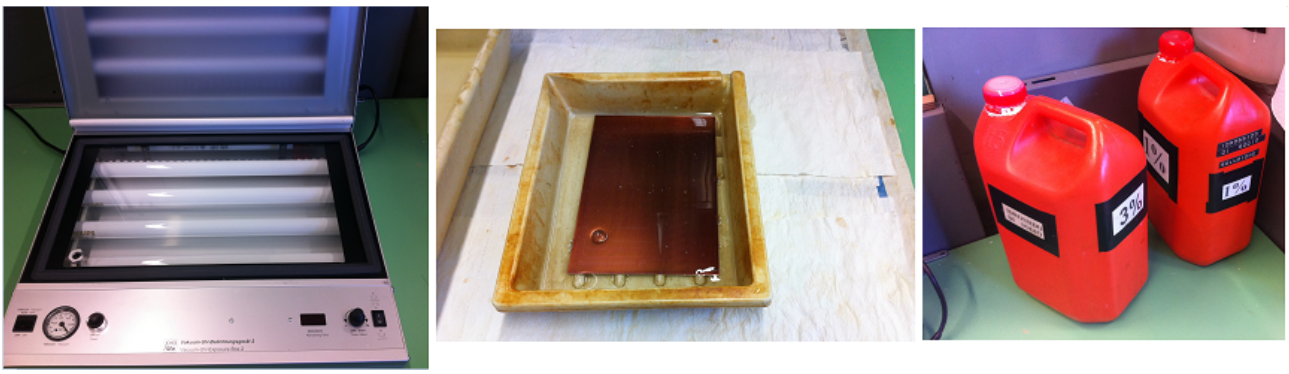


Fig.206

Lo schema PCB da utilizzare deve essere lato rame, visto che la basetta mono faccia non offre la possibilità di implementare il doppio strato, così come la strumentazione in dotazione, ad uso squisitamente hobbystico, non permette la creazione di schede elettroniche multi strato.

Il PCB visto dall'alto, quindi dal lato inserimento componenti, è quello di Fig.207, nel quale, oltre alle piste in blu, sono delineati i componenti elettrici con le rispettive sigle, i fori con le rispettive parti in rame, il bordo della scheda ed infine eventuali scritte personalizzate. La dimensione della figura non deve trarre in inganno visto che la dimensione originale del PCB in fase di stampa è pari a 6cmX5.5cm. Osservando la dimensione delle resistenze e del transistor BJT 337 si può avere una percezione della dimensione originale.

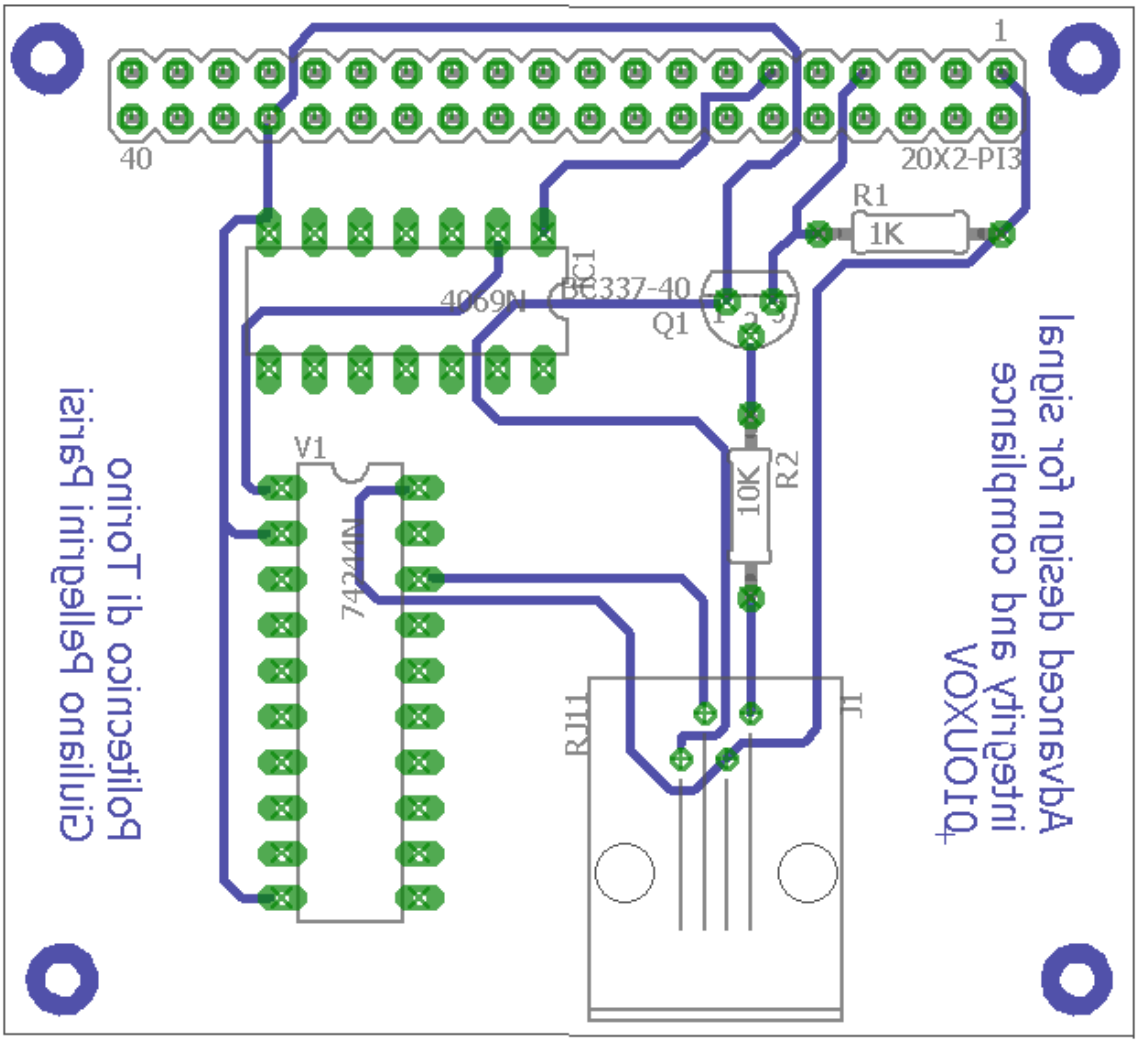


Fig.207

Il software Eagle permette facilmente il passaggio al lato rame, togliendo dalla visualizzazione le parti non necessarie, come ad esempio le componenti in grigio. La vista lato rame comprensiva di piste, pads e componenti è visibile in Fig.208.

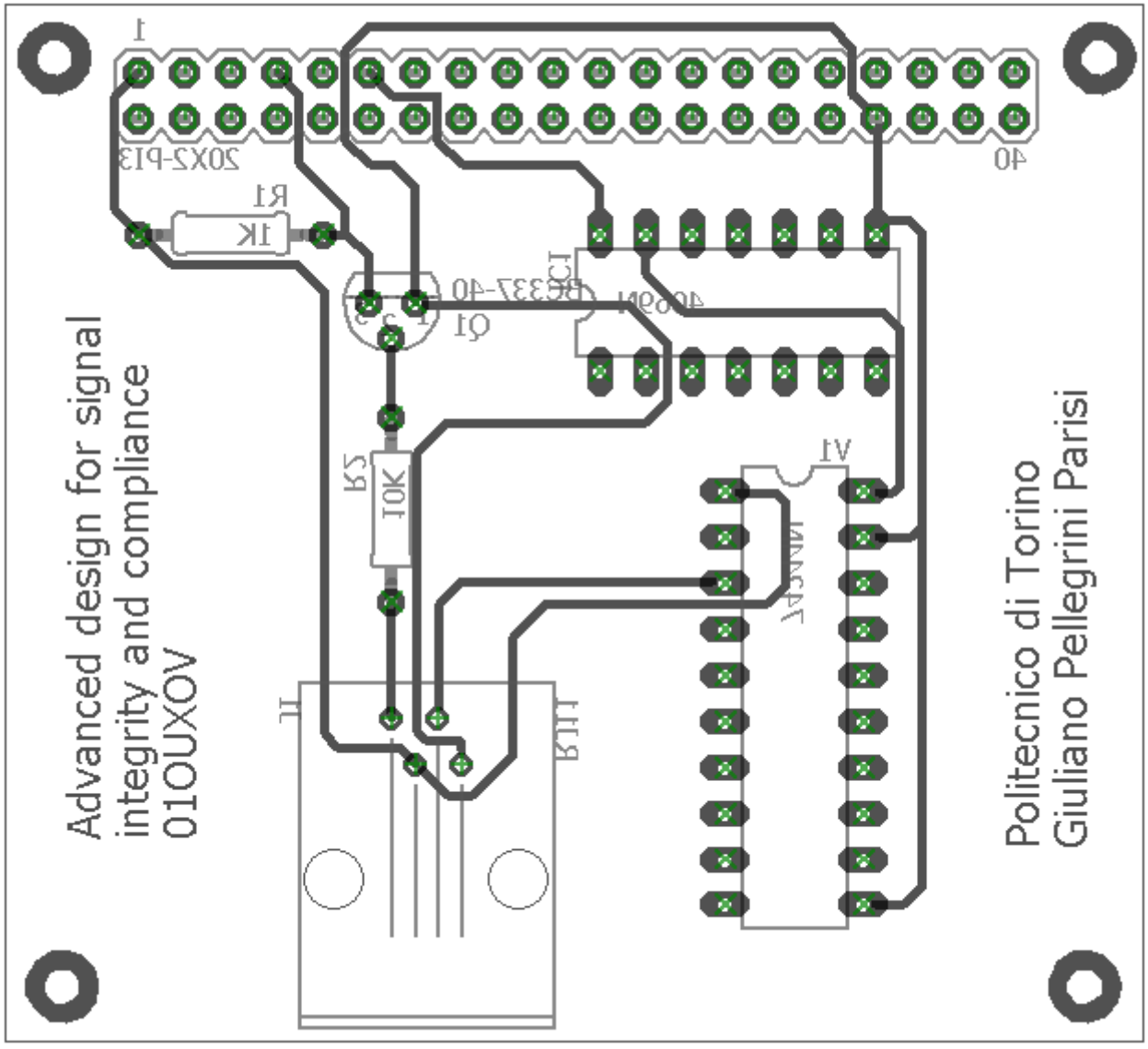


Fig.208

Lo schema PCB finale lato rame pronto per la stampa è quello di Fig.209.

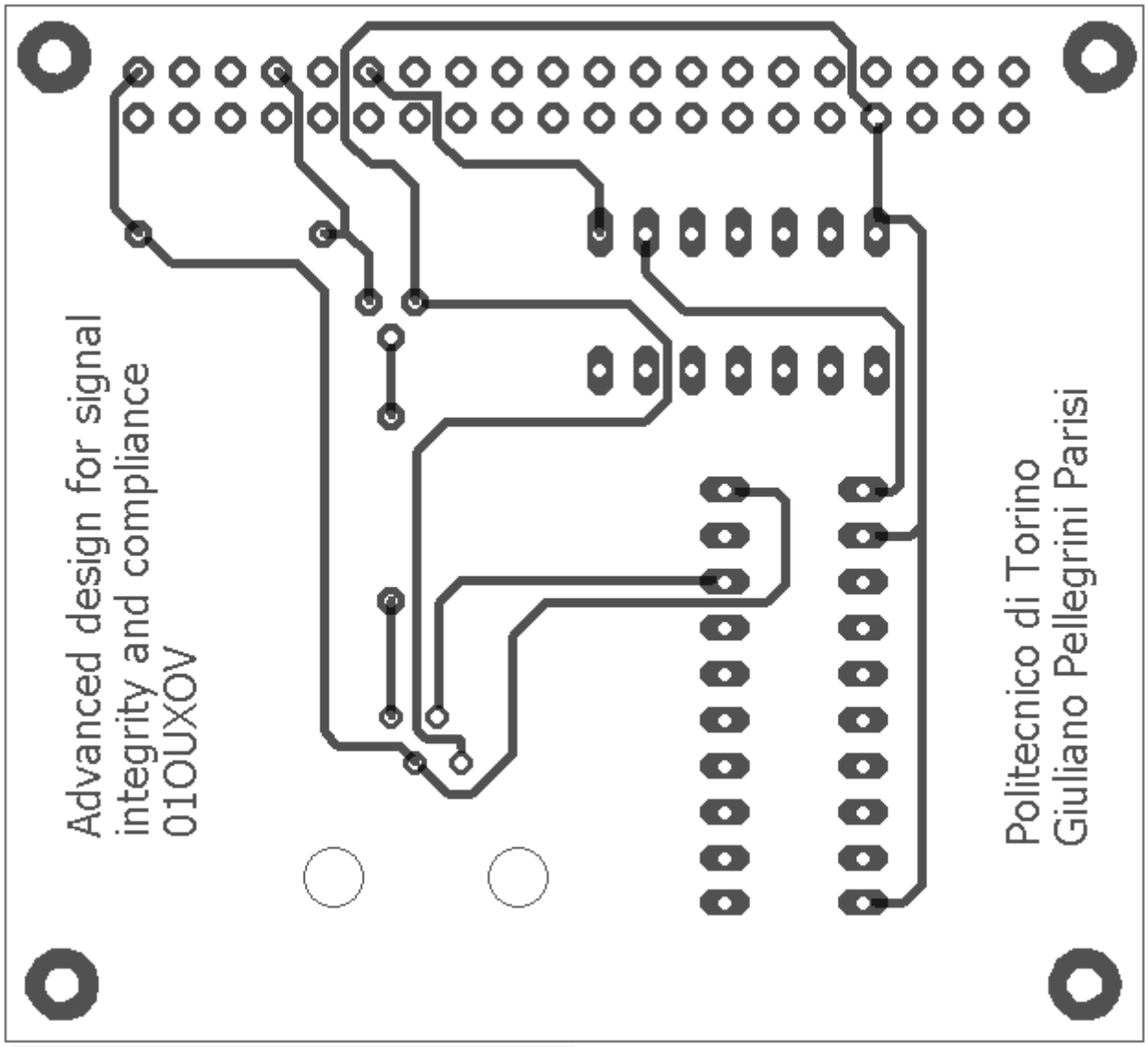


Fig.209

Realizzata fisicamente la scheda, non resta che passare all'operazione di foratura e installazione delle componenti elettroniche.

Nonostante la presenza della scheda, volendo effettuare il debug del segnale di trasmissione inverso tramite il DSO Rigol, si continuerà ad utilizzare la breadboard in fase di scrittura della UWP. Solo al termine del software con conseguente testing di funzionamento, si procederà ad utilizzare in modo esclusivo la scheda appena prodotta.

4.1.7 Test software per avvio datagramma

La parte elettrica di interfacciamento con l'anemometro TX20 è ormai pronta, resta da verificare se la corretta programmazione del GPIO17 del Pi 3 avvia la trasmissione dei dati e, una volta chiusa la UWP sul Raspberry, questa si interrompe. Come avvenuto con gli esempi introduttivi del capitolo 3, è necessario creare una UWP chiamata, per esempio, "TestTrasmissioneTX20". Si veda la Fig.210 e Fig.211 per la scelta del Build a 10240.

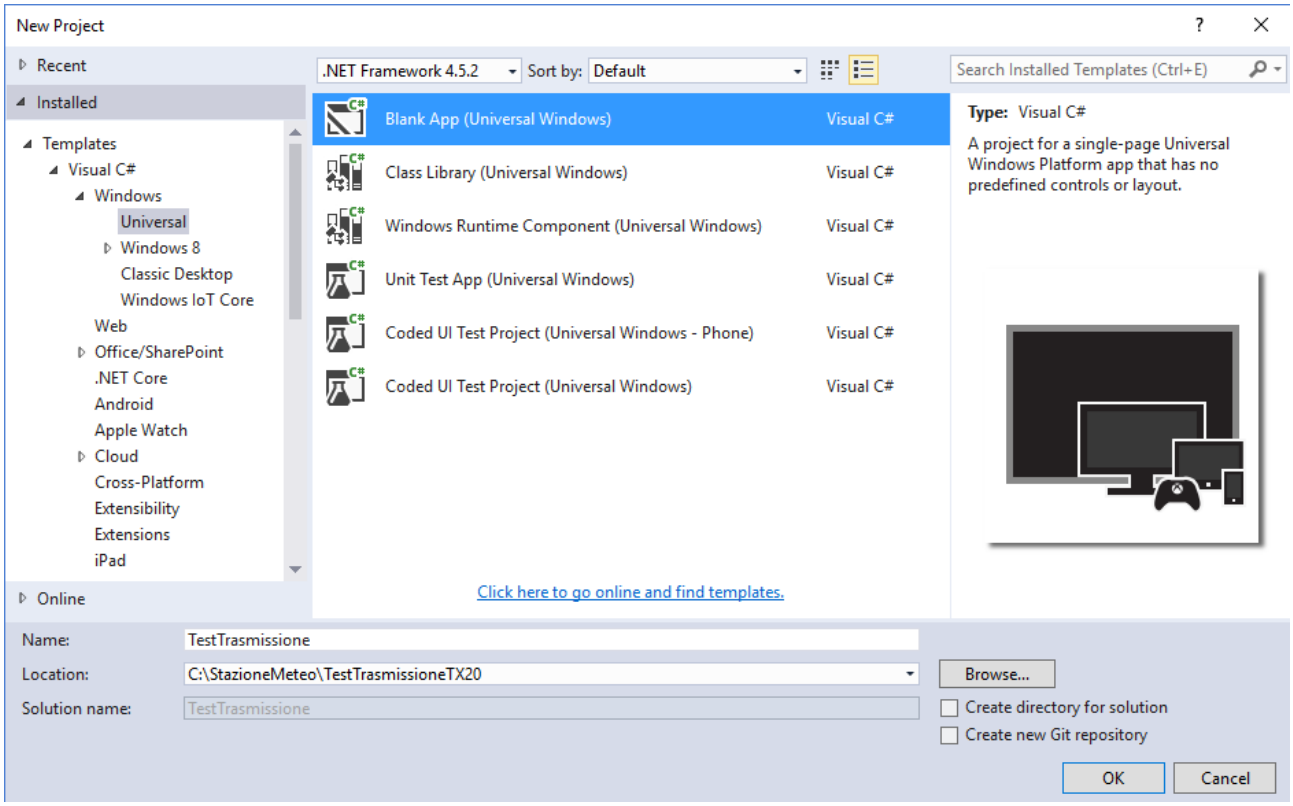


Fig.210

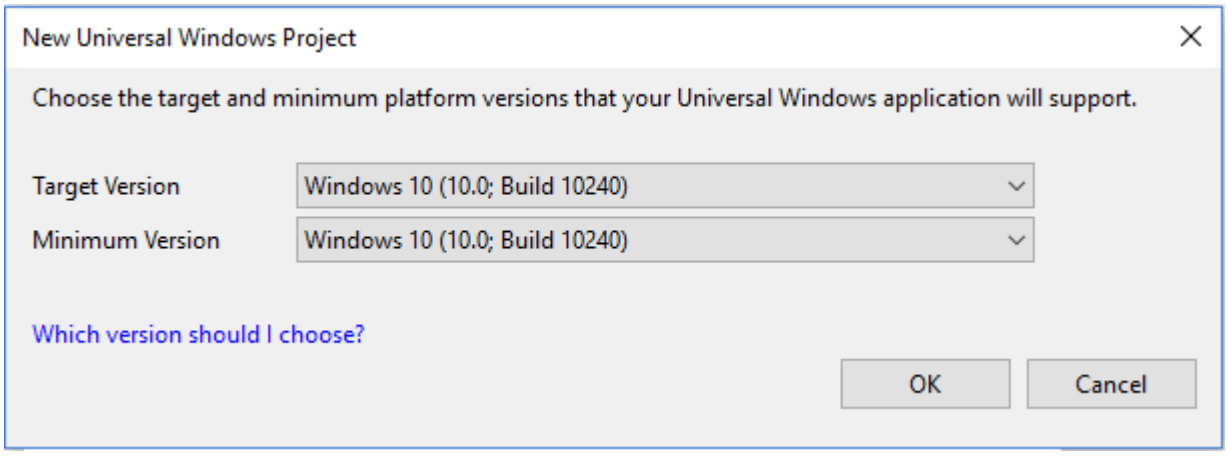


Fig.211

Per scrivere il codice C# si deve selezionare il file "MainPage.xaml.cs" dalla sezione "Solution Explorer" di Fig.212.

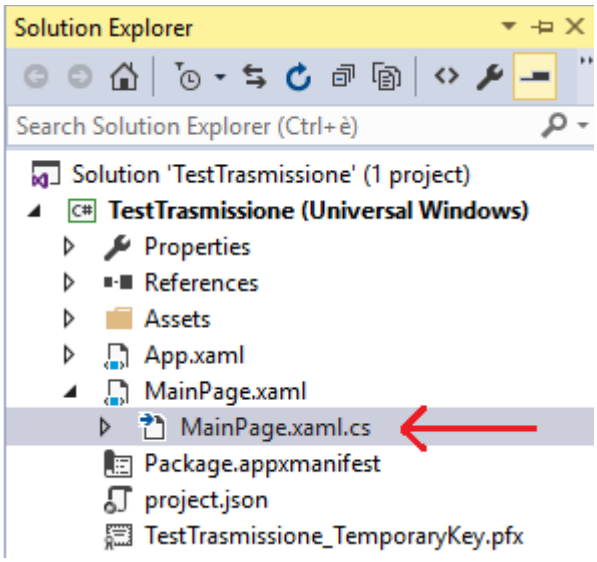


Fig.212

Successivamente è necessario agganciare la libreria per la gestione della GPIO che per default non è collegata al progetto. Nella "Solution Explorer" selezionare la voce "References" e poi, con il tasto destro del mouse, selezionare la voce "Add Reference...".

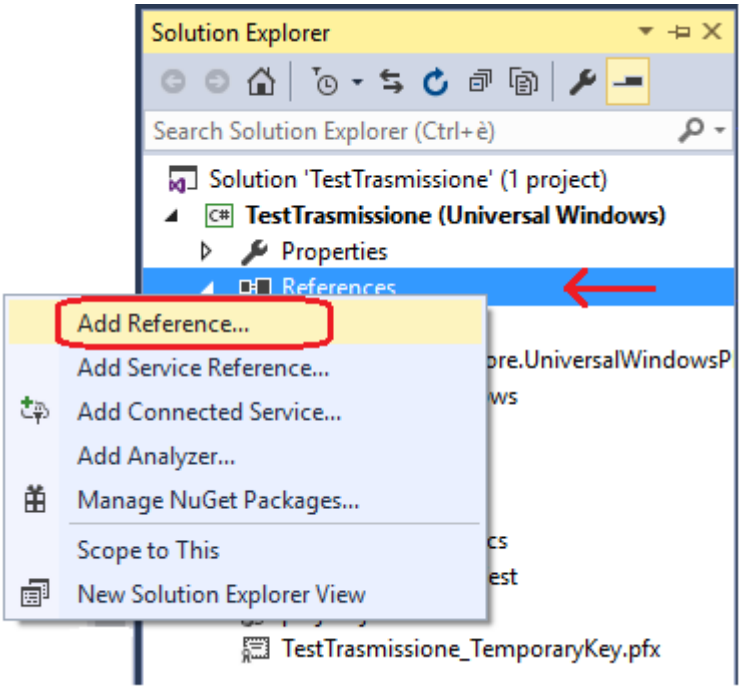


Fig.213

Selezionare nella sezione "Universal Windows" le librerie di estensione "Extensions" e poi selezionare la librerie "Windows IoT Extensions for the UWP" versione "10.0.10240.0". Confermare il tutto premendo il pulsante OK, come si vede in Fig.214. Successivamente la libreria viene aggiunta nella sezione "References". Si veda la Fig.215.

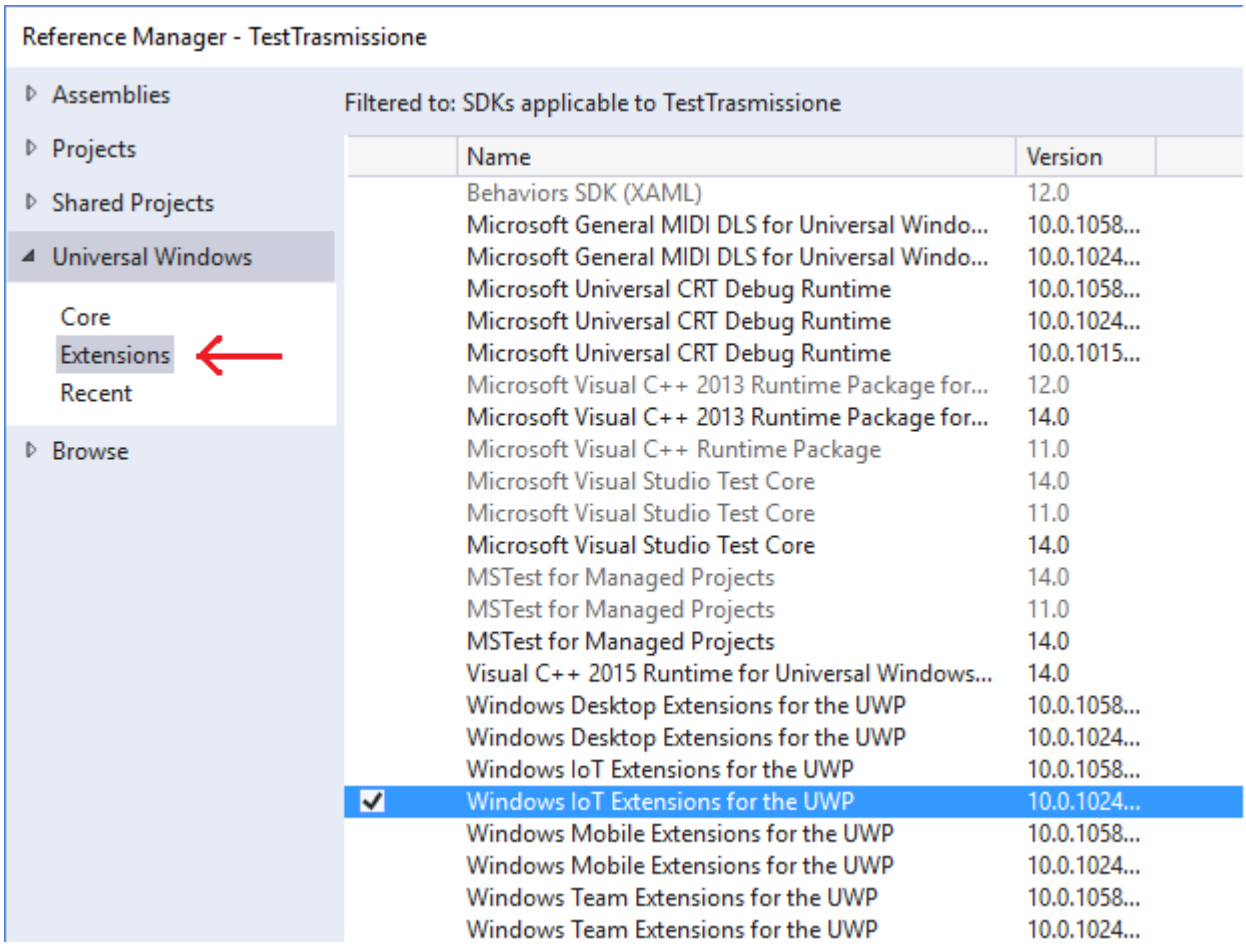


Fig.214

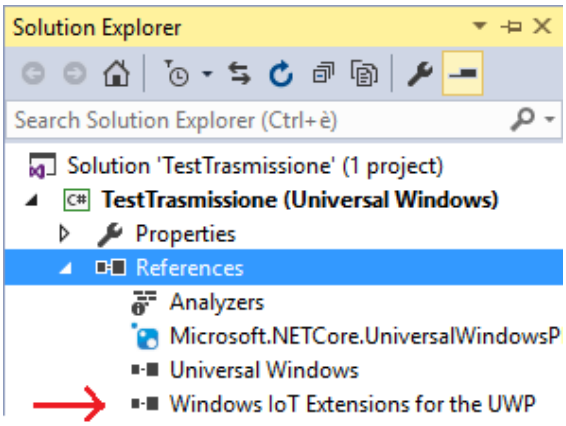


Fig.215

Riportare il codice C# sottostante nel file "MainPage.xaml.cs".

```
using Windows.UI.Xaml.Controls;
using Windows.Devices.Gpio;

namespace TestTrasmissione
{
    public sealed partial class MainPage : Page
    {
        private const int pinAttivazioneDTR = 17;
        public MainPage()
        {
            InitializeComponent();

            AttivaTX20(); // Avvia attivazione trasmissione automatica dei datagrammi
        }

        private void AttivaTX20()
        {
            var gpio = GpioController.GetDefault();

            if (gpio == null)
                return;
            else
            {
                var pinDTR = gpio.OpenPin(pinAttivazioneDTR);
                pinDTR.Write(GpioPinValue.High);
                pinDTR.SetDriveMode(GpioPinDriveMode.Output);
            }
        }
    }

}
```

Impostare l'architettura ARM per la UWP come in Fig.216 e, successivamente, impostare l'opzione "Remote Machine" per effettuare il Deployment remoto come da Fig.217.

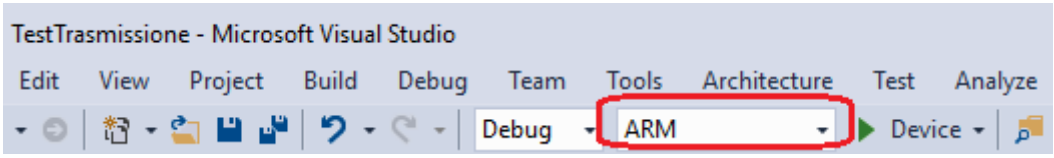


Fig.216

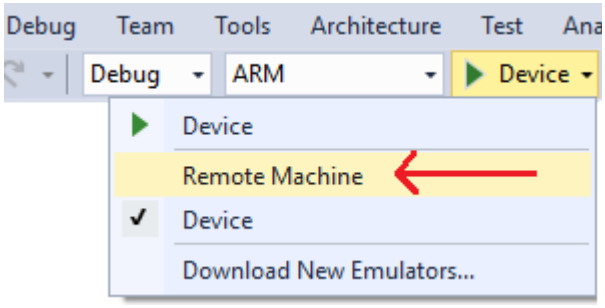


Fig.217

Impostare l'indirizzo IP nell'apposita casella ed eventualmente utilizzare il pulsante di ricerca per trovare una connessione, come indicato in Fig.218.

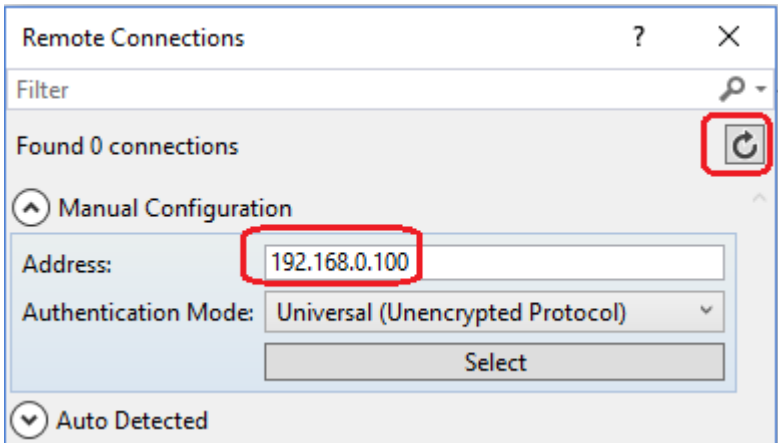


Fig.218

Procedere alla compilazione del progetto e alla sua installazione in remoto sul Pi 3, sfruttando la voce "Deploy Solution" del menù Build. Si veda la Fig.219.

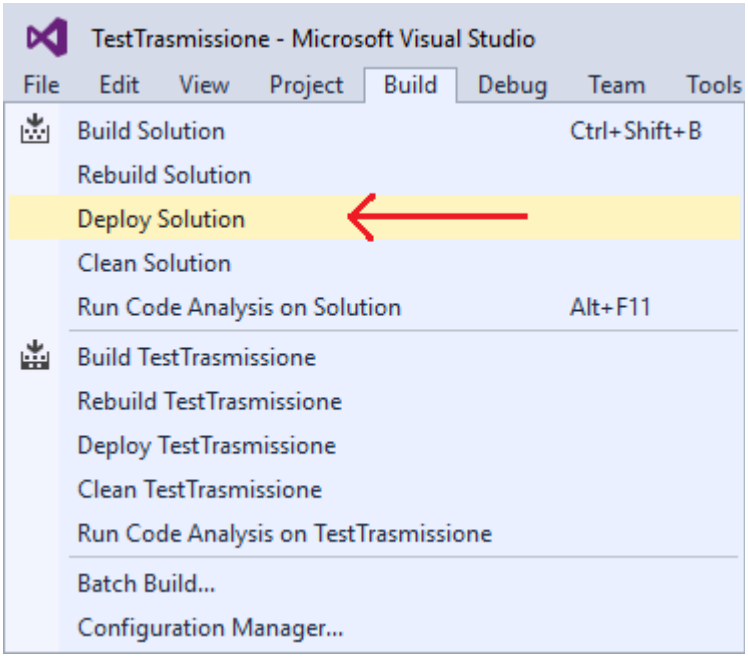


Fig.219

L'esito della compilazione e dell'installazione remota è visibile nella sezione di output come da Fig.220.

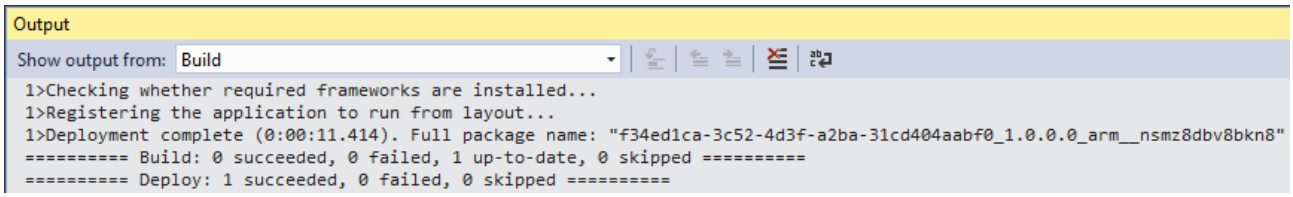


Fig.220

Lo step finale è accedere alla configurazione web del Raspberry, per fare ciò è possibile utilizzare la Dashboard la quale trova tutti i device correttamente configurati sulla rete. In modo equivalente è possibile accedere direttamente tramite il browser scrivendo la URL "http://<Indirizzo-IP>:8080", sostituendo l'indirizzo utilizzato. In Fig.221 si utilizza l'accesso tramite Dashboard.

Una volta trovato il dispositivo è necessario premere il tasto destro del mouse e selezionare la voce "Apri in Device Portal" con conseguente apertura del browser e richiesta delle credenziali di autenticazione. L'utente con i massimi privilegi è l'account "administrator". Si veda la Fig.222.

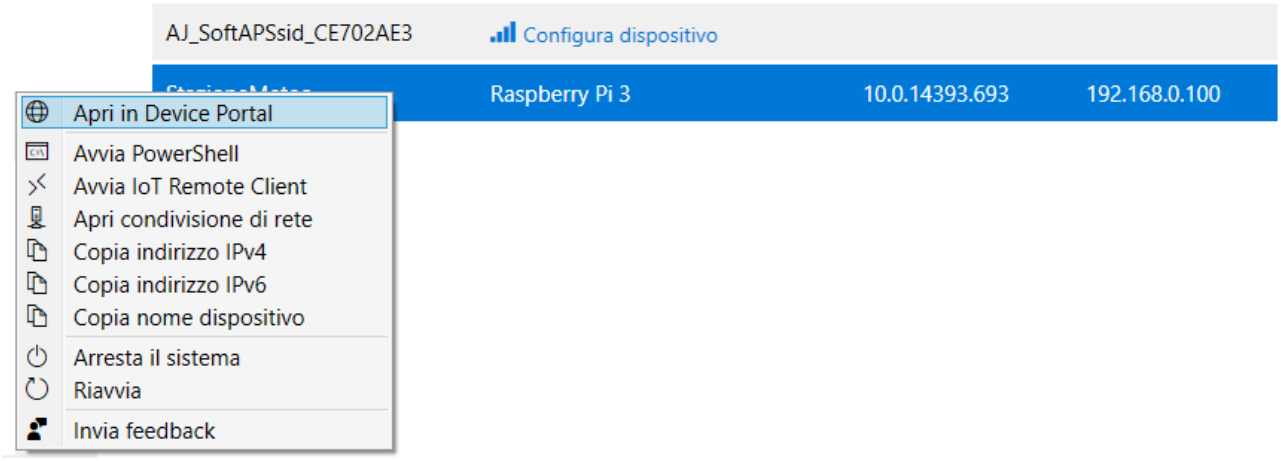


Fig.221

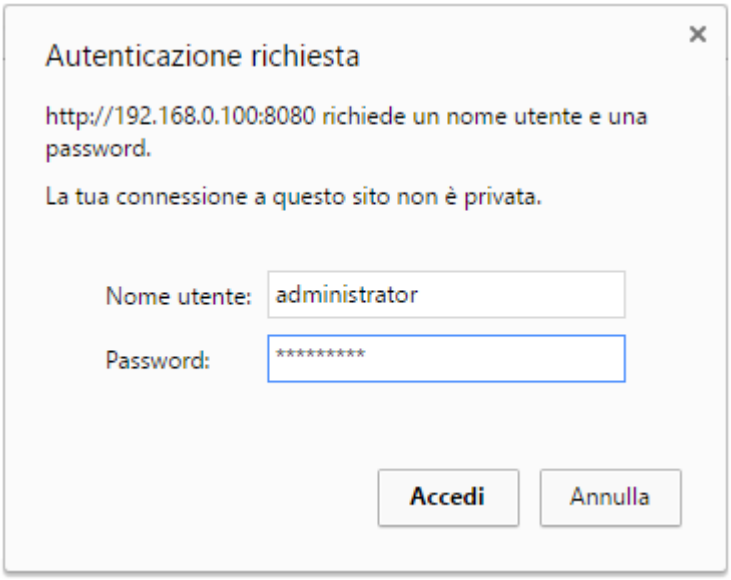


Fig.222

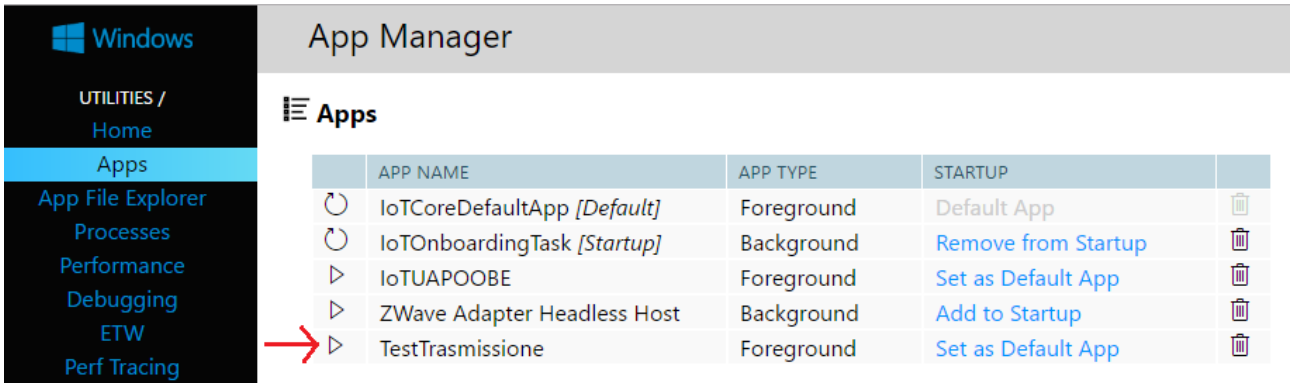


Fig.223

Nella sezione delle "Apps" deve essere presente la UWP appena installata, in caso contrario verificare la presenza di altri device Pi 3. Si veda la Fig.223. Prima di procedere all'avvio dell'applicazione "TestTrasmissione" con l'apposito pulsante triangolare, è bene verificare lo stato di trasmissione dei dati da parte del TX20, in questo modo è possibile avere la certezza del funzionamento dell'intero sistema, sia da un punto di vista hardware che software. Collegando il DSO direttamente al collettore del BJT, il segnale presente è quello di Fig.224 che risulta essere un livello alto visto la presenza della rete di inversione, che sta quindi a significare che l'anemometro non è in trasmissione e cmq il tempo di divisione di 1sec dimostra che non vi è nessun invio automatico ad ogni intervallo di 2sec.

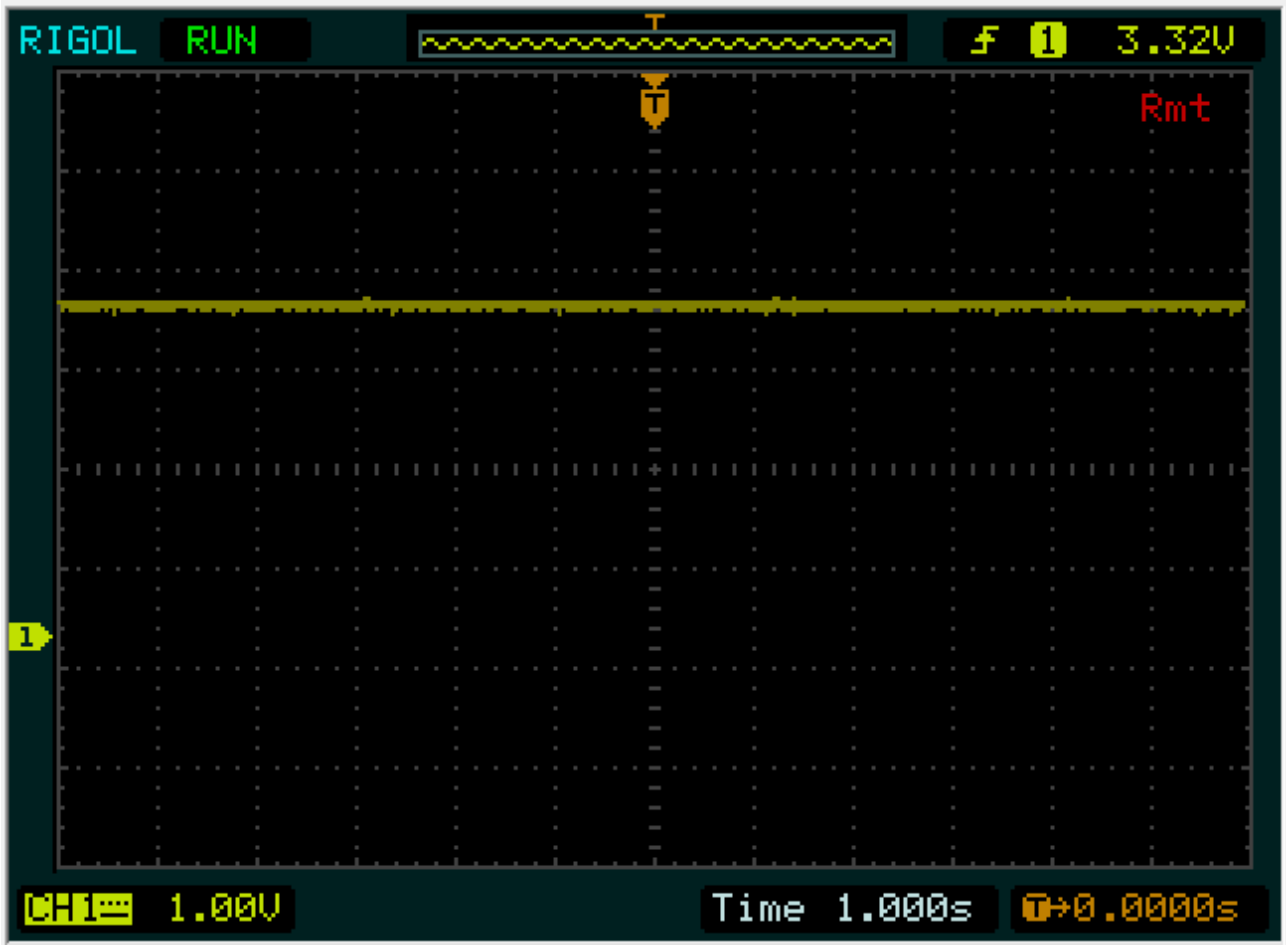


Fig.224

Il passo successivo è mandare in esecuzione la UWP "TestTrasmissione" ed osservare con attenzione il segnale al DSO per vedere se è presente o meno una sequenza di datagrammi mantenendo, per comodità, sempre 1sec come tempo di divisione. Si veda la Fig.225.

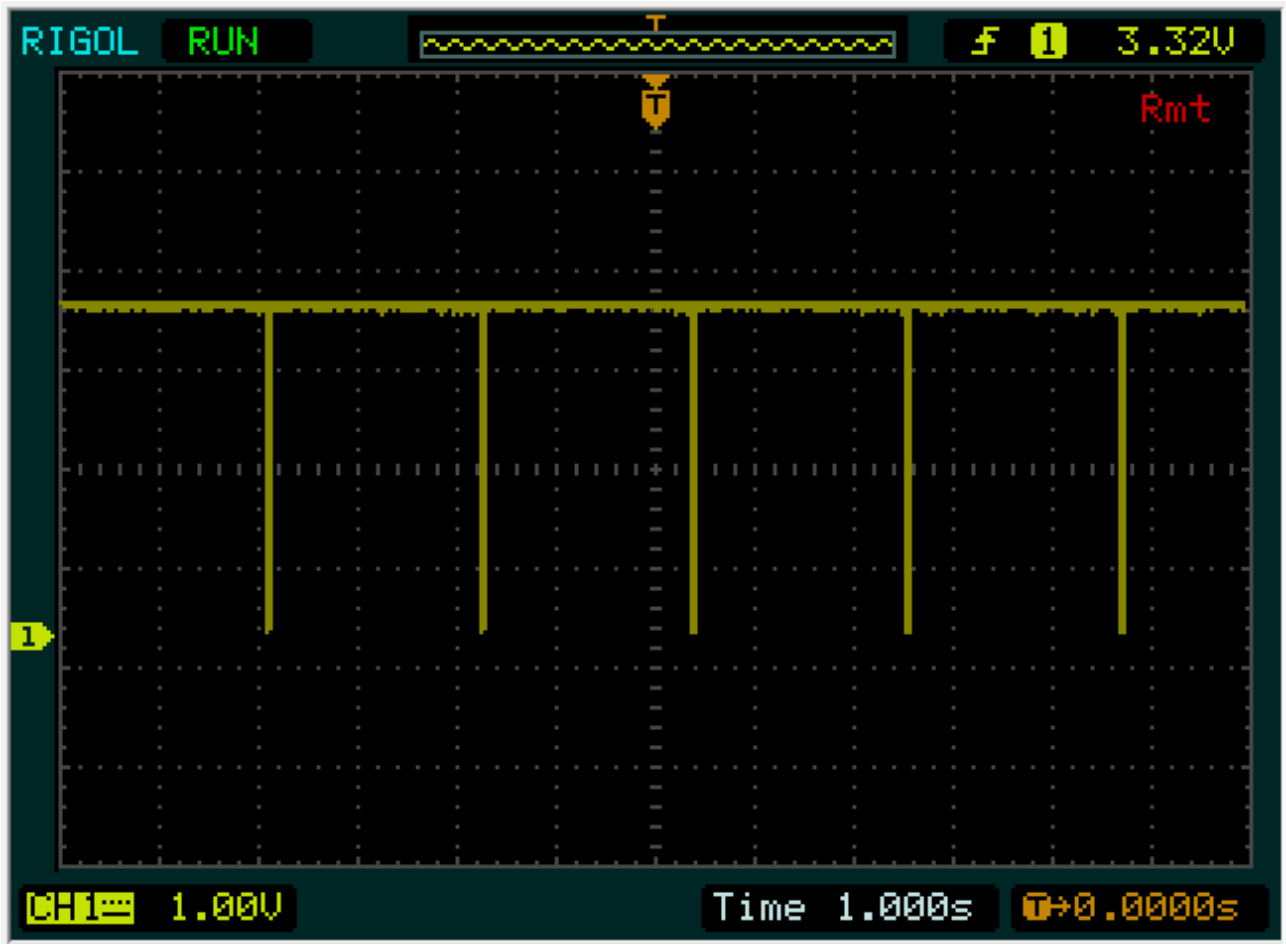


Fig.225

E' evidente l'invio di 5 datagrammi al Pi 3, di conseguenza l'applicazione software funziona in modo corretto. Non resta che verificare, alla chiusura della UWP tramite il pulsante stop di Fig.226, l'interruzione di qualsiasi attività di invio da parte dell'anemometro.

Fig.226

La Fig.227 riporta un comportamento del tutto analogo a quello antecedente l'avvio della UWP, questo dimostra che tutta la parte hardware ed il piccolo listato di codice C# funzionano in modo corretto.

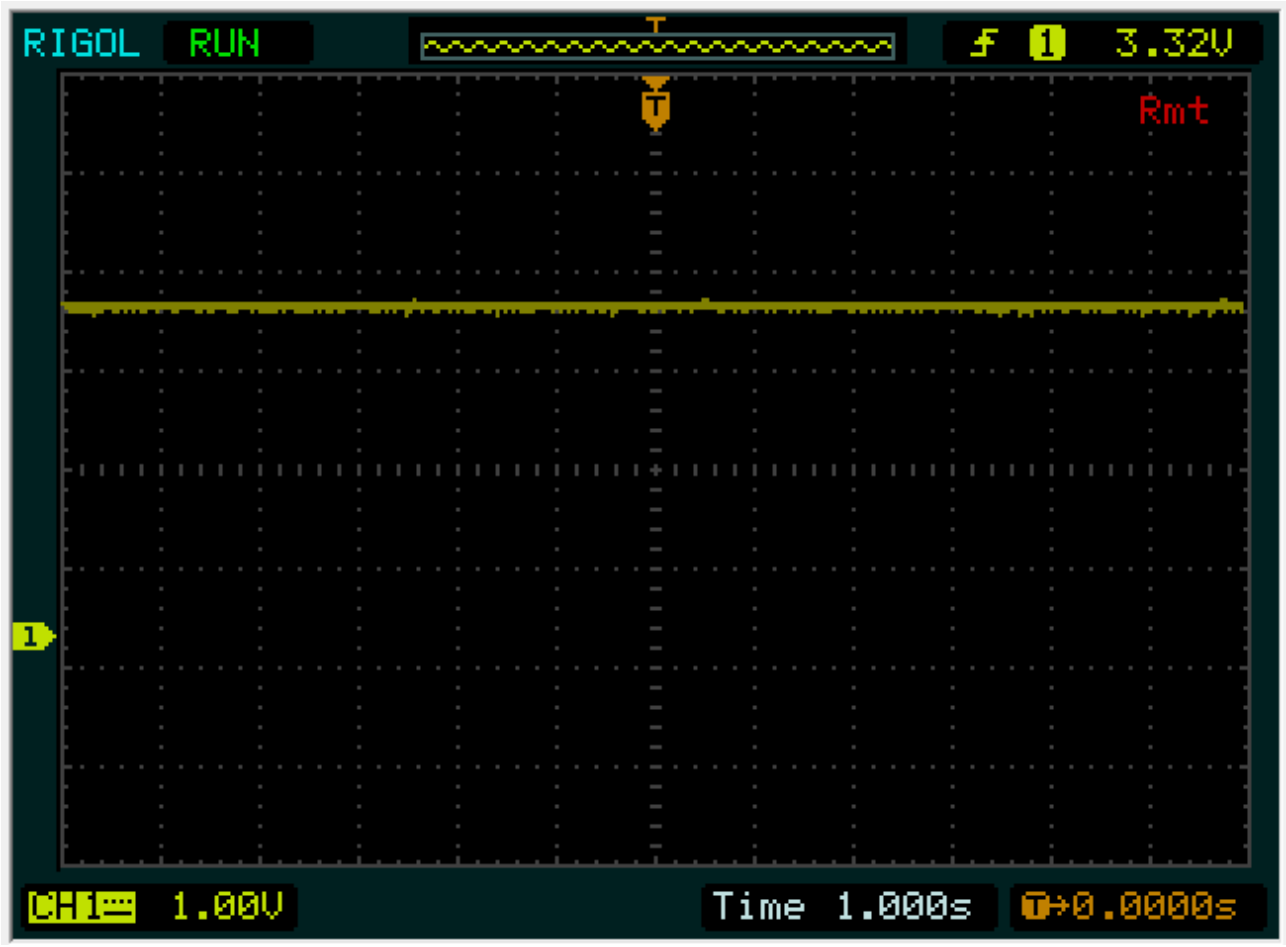


Fig.227

Questo paragrafo ha dimostrato che non basta collegare un dispositivo qualsiasi ai pin GPIO di una scheda embedded per avere la certezza della compatibilità elettrica tra i due device. Il più delle volte le apparecchiature commerciali che si vogliono interfacciare ad un Raspberry sono delle scatole chiuse con un livello di integrazione elettronico molto elevato, che ne comporta una difficile analisi per qualsiasi tentativo di reverse engineering. Il paragrafo successivo affronta la stesura del codice C# col fine di leggere correttamente tutti i bit del datagramma e produrre dei dati integri di velocità e direzione del vento.

4.2 UWP

La UWP che si vuole realizzare deve leggere i dati dall'anemometro in tempo reale e successivamente inviarli al Cloud Microsoft Azure. In questo paragrafo viene presentata la UWP per la lettura dei dati, con una prima analisi dell'algoritmo, così da visualizzare i dati sia a monitor sia nella console di debug. La trattazione del cloud sarà oggetto del capitolo successivo.

4.2.1 Reverse engineering

Prima di scrivere una sola riga di codice C#, è bene analizzare tramite diagramma di flusso come imbastire l'algoritmo, così da poterlo tradurre anche in altri linguaggi di programmazione, come ad esempio il C++.
E' bene ricordare quanto estrapolato dal reverse engineering del paragrafo 4.1.3, passi che vengono qui riassunti ricordando che è presente l'inversione dei bit hardware.

1. Lettura dei primi 5 bit di start come sequenza di 11011
2. Lettura dei successivi 4 bit ad indicare la direzione del vento
3. Lettura dei successivi 12 bit ad indicare la velocità del vento
4. Lettura dei successivi 4 bit ad indicare la parte di checksum
5. Lettura dei successivi 4 bit ad indicare la ripetizione della direzione del vento. Tali bit nella logica di trasmissione del TX20 vengono inviati non invertiti
6. Lettura dei successivi 12 bit ad indicare la ripetizione della velocità del vento. Tali bit nella logica di trasmissione del TX20 vengono inviati non invertiti
7. La lunghezza del singolo bit è pari a 1220μsec
8. E' necessario che il checksum sia uguale ai bit della direzione e velocità del vento inviati invertiti, secondo il seguente schema:

<direzione-vento-corretta>: <4 bits>
<velocità-vento-corretta>: <bit da 1 a 4>
<velocità-vento-corretta>: <bit da 5 a 8>
<velocità-vento-corretta>: <bit da 9 a 12>

Risultato: <SOMMA bit a bit> = <checksum-corretto> ?

4.2.2 MainPage.xaml

La parte grafica della UWP viene realizzata in modo semplice con un solo componente "RelativePanel" e quattro "TextBlock" per la visualizzazione della direzione e velocità del vento nelle unità di misura m/s, km/h e mph. In qualsiasi momento è possibile arricchire la parte grafica con oggetti visuali più accattivanti, parte che al momento esula da tale trattazione, anche perché il Raspberry verrà poi installato privo di periferiche di visualizzazione. La Fig.228 mostra la semplice realizzazione su monitor 48" utilizzato in fase di testing.

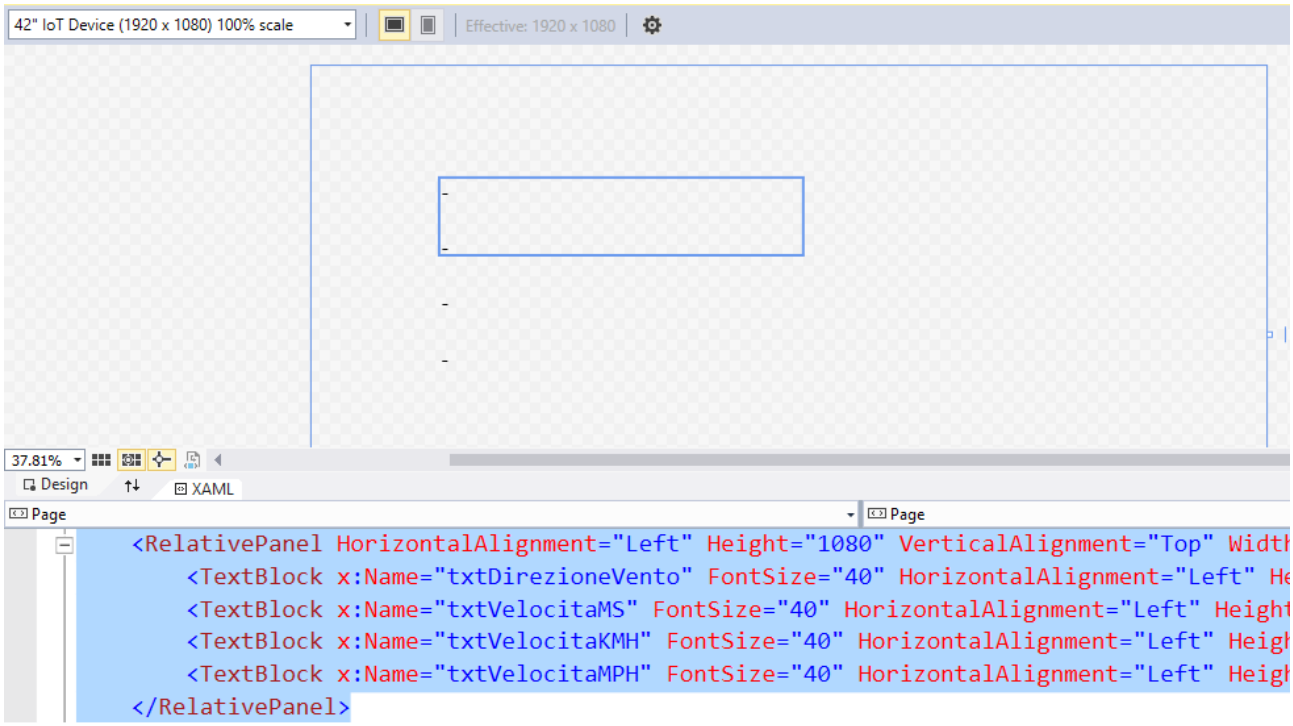


Fig.228

Le componenti "TextBlock" sono inserite all'interno del contenitore "RelativePanel" ed inizializzate con un carattere a scelta, tipo "-". La dimensione del font è pari a 40, mentre i nomi dei componenti deve essere diverso, come si evince dalla figura. Tali nomi verranno poi utilizzati per inviare il rispettivo dato da visualizzare. Il programmatore può lavorare direttamente a livello di codice XAML, il quale permette una più semplice impostazione di tutta la parte grafica. Segue il codice XAML completo della parte GUI.

```
<Page
    x:Class="StazioneMeteo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:StazioneMeteo"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
   <RelativePanel HorizontalAlignment="Left" Height="1080" VerticalAlignment="Top"
                  Width="1920">
        <TextBlock x:Name="txtDirezioneVento" FontSize="40" HorizontalAlignment="Left"
                   Height="155" TextWrapping="Wrap" Text="-"
                   VerticalAlignment="Top"  Width="725" Margin="260,225,-923,-360"
                   RenderTransformOrigin="0.968,1.5"/>
        <TextBlock x:Name="txtVelocitaMS" FontSize="40" HorizontalAlignment="Left"
                   Height="155" TextWrapping="Wrap" Text="-"
                   VerticalAlignment="Top" Width="255"
                   RenderTransformOrigin="1.62,5.156" />
       <TextBlock x:Name="txtVelocitaKMH" FontSize="40" HorizontalAlignment="Left"
                  Height="155" TextWrapping="Wrap" Text="-"
                  VerticalAlignment="Top" Width="255"
                  RenderTransformOrigin="1.62,5.156" Margin="260,449,-444,-572"/>
       <TextBlock x:Name="txtVelocitaMPH" FontSize="40" HorizontalAlignment="Left"
                  Height="155" TextWrapping="Wrap" Text="-"
                  VerticalAlignment="Top" Width="255"
                  RenderTransformOrigin="1.62,5.156" Margin="260,561,-444,-684"/>
    </RelativePanel>
</Page>
```

4.2.3 MainPage.xaml.cs

La parte di codice C# contenuta nel file "MainPage.xaml.cs" è divisa in 12 metodi così da rendere il codice più flessibile e di facile lettura. L'intero programma utilizza ovviamente classi e metodi del .NET Core, anche se la logica di realizzazione non utilizza il paradigma ad oggetti. Nell'elenco puntato che segue viene indicato il nome e lo scopo dei relativi metodi.

- MainPage = metodo di base avviato automaticamente dalla UWP
- attivazioneTX20 = metodo che avvia la trasmissione dei dati dal TX20 lavorando sul GPIO17
- letturaDatiDaTX20 = metodo che rileva lo start frame ed estrapola i campi direzione e velocità, visualizzandoli sia a console che a video
- inizializzazioneGPIO = metodo che rileva la presenza della GPIO ed imposta la GPIO4
- startFrame = metodo per la rilevazione dei 5 bit di start
- direzioneVento1 = metodo per la rilevazione dei 4 bit di direzione del vento
- direzioneVento2 = metodo per la rilevazione dei 4 bit di direzione del vento nel quale però avviene l'inversione logica, essendo tale campo spedito non invertito da parte del TX20
- velocitaVento1 = metodo per la rilevazione dei 12 bit della velocità del vento
- velocitaVento2 = metodo per la rilevazione dei 12 bit della velocità del vento nel quale però avviene l'inversione logica, essendo tale campo spedito non invertito da parte del TX20
- checksum = metodo per la rilevazione dei 4 bit di checksum
- calcoloDelChecksumSulDatagramma = metodo per la comparazione dei bit di ckecksum con i bit di direzione e velocità del vento della prima trasmissione
- delay = metodo per la realizzazione di un ritardo bloccante in micro secondi

Conviene vedere nel dettaglio analisi ed implementazione del codice, così da capire le scelte proposte dall'autore ed eventuali migliorie, che in ambito software sono sempre possibili. Il paragrafo 4.2.5 riporterà l'intero codice del programma C#.

4.2.3.1 MainPage()

Questo metodo è quello di default creato dall'ambiente Visual Studio, ed è il punto di avvio automatico di tutta l'applicazione UWP. Il diagramma di flusso di Fig.229 riporta la semplice implementazione di tale metodo, che in definitiva si limita ad eseguire il metodo "attivazioneTX20()".

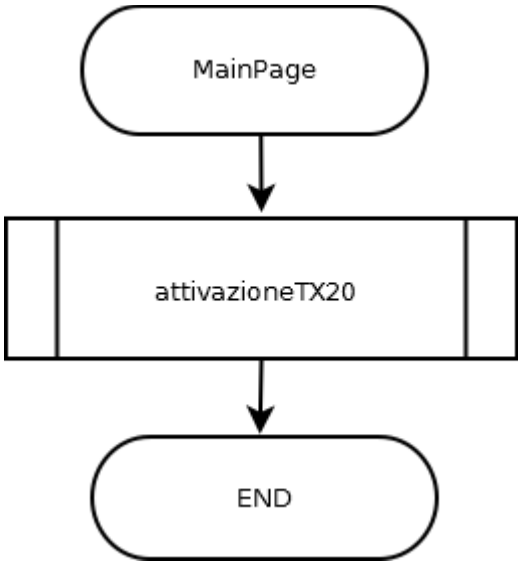


Fig.229

Il codice C# implementativo è veramente immediato.

```
public MainPage()
{
    this.InitializeComponent();

    attivazioneTX20();
}
```

E' necessario eseguire il metodo "attivazioneTX20()" dopo l'inizializzazione di tutta la parte grafica della UWP. Questo step viene fatto tramite il metodo di classe "InitializeComponent()" già presente nel progetto.

4.2.3.2 attivazioneTX20()

Questo metodo serve ad inizializzare la GPIO e qualora non sia presente sul device embedded l'intera applicazione non potrà ovviamente venire eseguita. Successivamente è necessario attivare l'anemometro TX20 per la trasmissione, quindi si deve programmare la GPIO17 in uscita con stato logico alto. Il flow chart di Fig.230 riassume la logica implementativa.

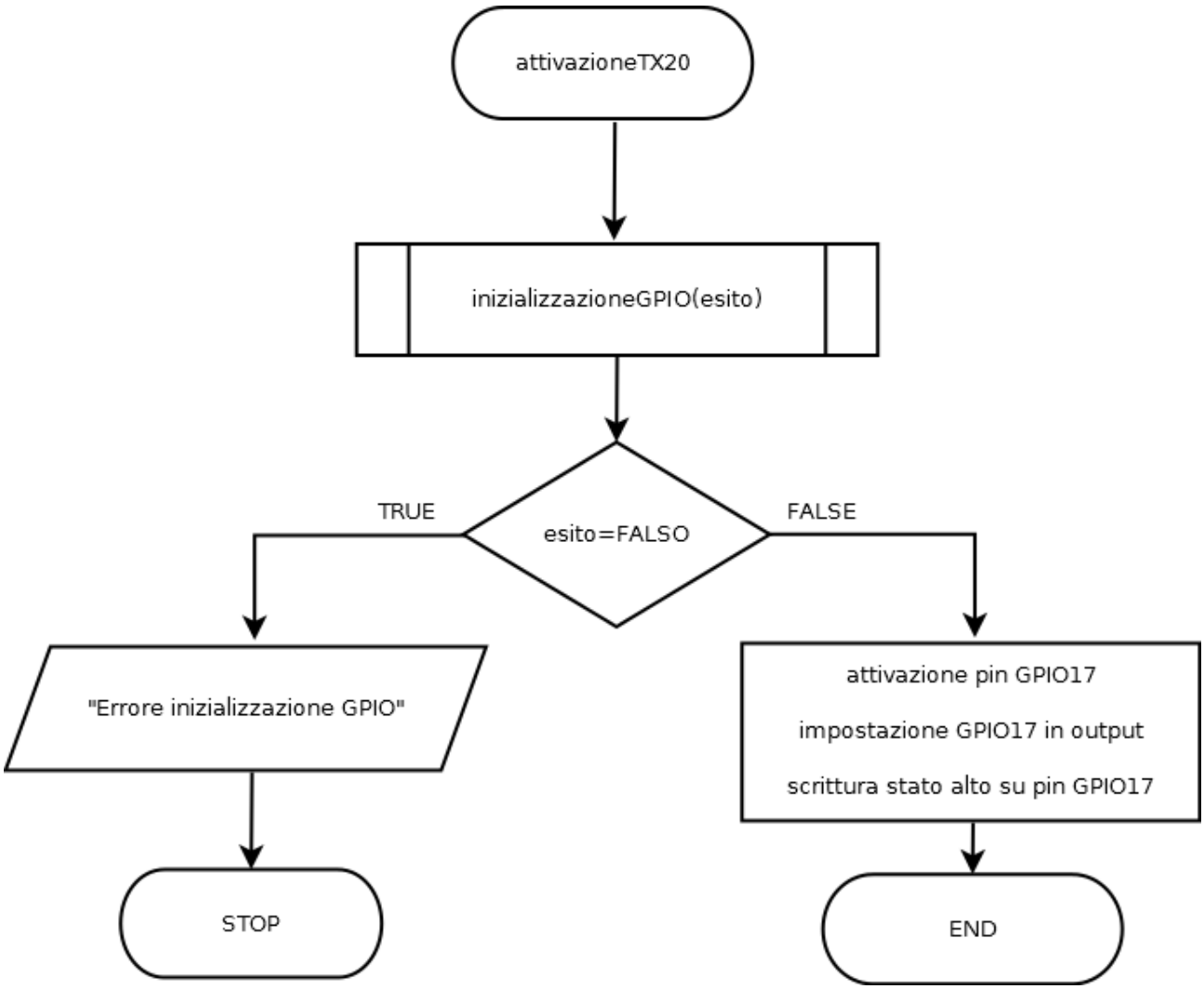


Fig.230

Tale metodo richiama la funzione "inizializzazioneGPIO()" che restituisce un valore booleano sulla base o meno della presenza della GPIO sulla scheda. In caso negativo viene stampato in una "TextBlock" un messaggio ed interrotta l'esecuzione della UWP. In

caso contrario è necessario procedere all'inizializzazione della GPIO17 e alla sua successiva configurazione, impostando il pin in uscita e con stato logico alto così che il chip HCF4069 ponga in uscita sul suo pin numero 2 lo stato logico basso, necessario a fare scattare il 3-state del 74LS244 il quale, di conseguenza, riporta lo stato logico basso collegato direttamente al pin numero 2, sull'uscita 18 a cui è collegato il segnale DTR dell'anemometro.
Segue il codice C# di tale metodo.

```
public sealed partial class MainPage : Page
{
    private const int pinAttivazioneTrasmissioneTX20 = 17;
    ...
    ...
    private void attivazioneTX20()
    {
        if (inizializzazioneGPIO()==false)
        {
            txtDirezioneVento.Text = "Errore inizializzazione GPIO!";

            return;
        }

        pinGPIO17 = gpio.OpenPin(pinAttivazioneTrasmissioneTX20);
        pinGPIO17.SetDriveMode(GpioPinDriveMode.Output);
        pinGPIO17.Write(GpioPinValue.High);
    }
```

Viene riportato anche il dato membro di classe "pinAttivazioneTrasmissioneTX20" così da evitare dubbi sul significato dello stesso all'interno del metodo, prassi non sempre seguita in molte documentazioni tecniche.

4.2.3.3 inizializzazioneGPIO()

Questo metodo esegue l'inizializzazione del controllore della GPIO, restituendo un valore booleano "false" qualora la stessa GPIO non sia presente sul device. In caso contrario si procede a configurare la GPIO4 necessaria a leggere i campi del datagramma spedito dal TX20. Questo pin deve quindi venire configurato in ingresso e deve essere accompagnato da un metodo delegate invocato automaticamente nel momento in cui vi è una variazione di tensione dallo stato logico alto a basso. E' possibile pensare tale delegate come ad un interrupt software in ambito .NET.

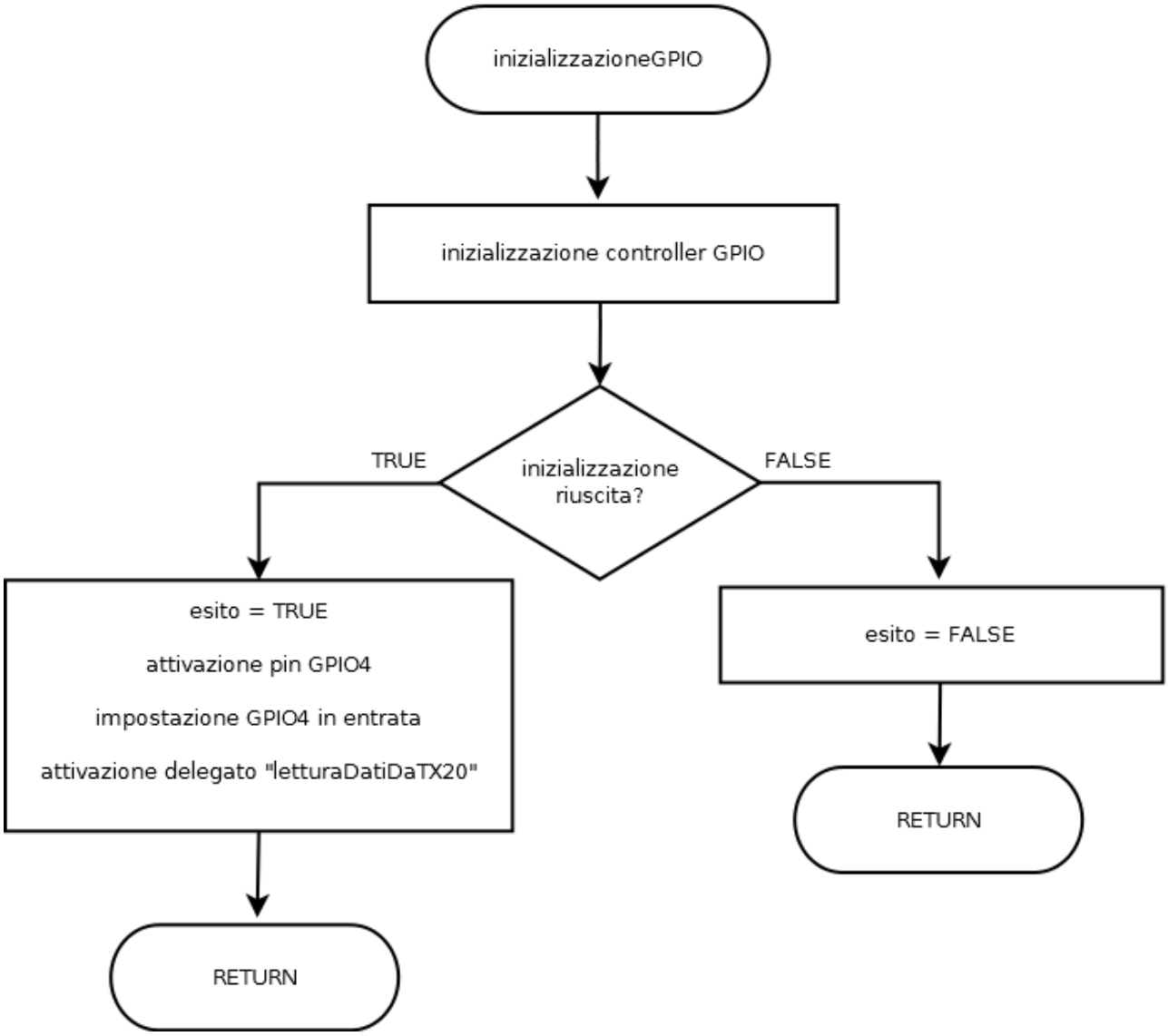


Fig.231

La parte di codice per la gestione del delegate segue, sulla falsa riga quanto visto negli esempi introduttivi dal capitolo 3, l'impiego del metodo "ValueChanged" del pin GPIO4 in ingresso. Per essere più precisi, ogni variazione logica rilevata su tale pin fa scattare la chiamata al metodo "letturaDatiDalTX20()", quindi anche una variazione da stato logico basso ad alto. Sarà necessario di conseguenza, al rilevamento della transizione da stato alto a basso in presenza dello start frame, disattivare la chiamata del delegate, tramite l'uso dell'operatore "-=". Questo aspetto sarà rivisto nell'apposito metodo "letturaDatiDalTX20()".

```
private bool inizializzazioneGPIO()
{
    this.gpio = GpioController.GetDefault();

    if (gpio == null)
        return false;
    else
    {
        pinGPIO4=gpio.OpenPin(pinRicezioneTX20);

        pinGPIO4.SetDriveMode(GpioPinDriveMode.Input);

        pinGPIO4.ValueChanged += letturaDatiDaTX20;

        return true;
    }
}
```

Quando il TX20 non trasmette, il segnale di uscita TxD è posto a 0V, ma per la presenza della rete di inversione, il pin GPIO4 rileverà una tensione di +3.3V, ossia stato logico alto. Nel momento in cui il DTR si trova a 0V, ossia stato logico basso, sul TxD viene inviato il datagramma, la cui intestazione è lo start frame 11011, ma grazie all'inversione hardware, la sequenza diviene 00100. La UWP resta quindi in attesa che il segnale cambi il suo stato logico da alto (nessuna trasmissione) a basso, primo bit dello start frame.

4.2.3.4 letturaDatiDalTX20()

Questo metodo è il delegate avviato in automaticamente ad ogni variazione di stato. Il diagramma di flusso di questa funzione è stato diviso in due parti a causa della sua lunghezza. La Fig.232 riporta la prima parte del flow chart nella quale si verifica inizialmente la rilevazione della variazione di transizione da stato logico alto a basso.

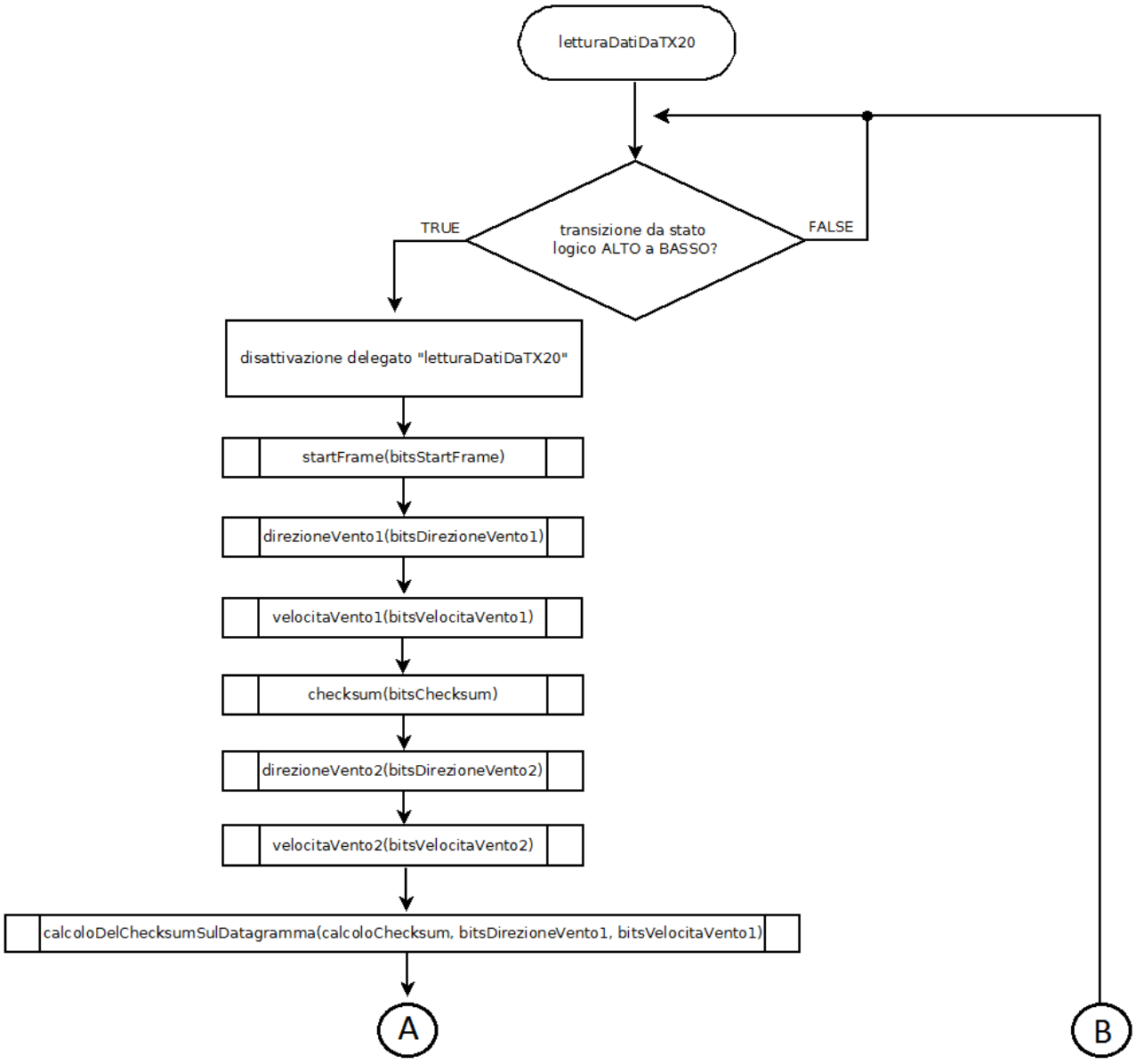


Fig.232

Questo test non avviene in polling, ma solo quando il pin rileva una qualsiasi variazione di stato. Il polling può essere facilmente implementato, ma sconveniente perché ha il grosso

difetto di occupare cicli di CPU inutilmente. La soluzione basata sull'interruzione è nettamente più ottimizzata, anche se parlare in un ambiente gestito come C#, che utilizza .NET Core di interrupt è alquanto fuori luogo. Il polling comunque non è la madre di tutti i mali, anzi, in questi ultimi anni con l'avvento delle architetture multi core, tale tecnica di facile realizzazione è stata ampiamente utilizzata nei driver di rilevamento dei pacchetti di rete, col fine di realizzare software di packet filtering. Alcuni firewall software, per avere la certezza di rilevare e permettere il corretto processamento di tutti i pacchetti, dedicano una core della CPU all'analisi in polling del traffico. Questa tematica è assai vasta, ma l'importante è conoscere e capire quale approccio utilizzare quando si vuole leggere dati dall'esterno. In questo sistema, visto l'intervallo di 2 secondi tra un datagramma e l'altro, l'uso del polling si rileva essere un inutile spreco di tempo di CPU.

Una volta rilevata la transizione da stato logico alto a basso, è necessario togliere l'unica chiamata a tale delegate, che altrimenti provocherebbe, in presenza di una seconda, terza o successiva generica transizione, l'invocazione continua della funzione "letturaDatiDalTX20()" in modo continuo nel tempo. Ovviamente tale situazione non potrebbe portare ad un'analisi corretta dei dati, motivo per il quale la chiamata di questo delegate viene temporaneamente disabilitata. Volendo sempre trovare una corrispondenza con le interrupt, è come procedere alla disabilitazione dell'interruzione una volta che si è entrati nel codice della ISR (Interrupt Service Routine). Disabilitato il delegate, si richiamano in sequenza i 7 metodi per il rilevamento dei campi, partendo proprio dello start frame, visto che è necessario leggere i 5 bit corrispondenti prima di potere processare i campi che portano informazioni utili. Le successive chiamate sono autoesplicative, ed estrapolano i rispettivi campi. Prima di procedere a visualizzare il dato, è obbligatorio verificare l'integrità dei dati, verificando la corrispondenza del campo checksum con i bit, opportunamente manipolati, dei campi direzione e velocità del vento prima trasmissione. Qualora il risultato non sia conforme, siamo in presenza di informazioni non valide, quindi non si deve assolutamente visualizzare nulla a video. La Fig.233 riporta i vari test di verifica, che ovviamente potevano venire riassunti in un unico controllo condizionale, ma che a livello di diagramma di flusso sarebbe stato imponente graficamente. Il primo test logico da farsi è il rilevamento corretto dello start frame, visto che il TX20 invia 11011, ma che per la presenza dell'inversione hardware diviene 00100, ossia il numero 4 decimale. Se il test fallisce non ha senso proseguire nell'analisi dei dati,

quindi si inserisce un ritardo simbolico di 1 secondo riattivando il delegate per procedere all'analisi di un successivo datagramma.

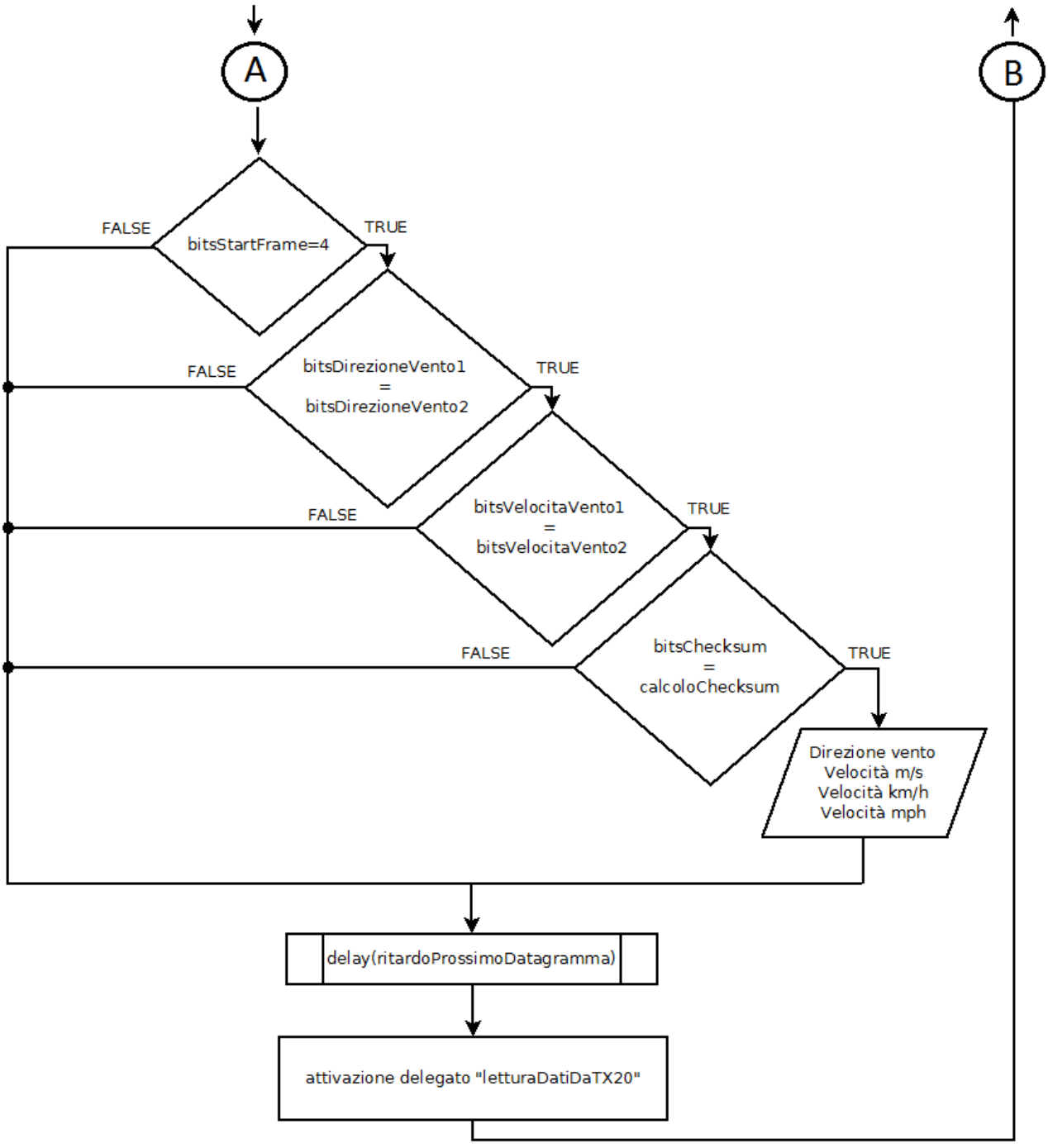


Fig.233

Qualora lo start frame sia corretto, i bit dei due campi che identificano la direzione del vento, devono ovviamente essere corretti. E' importante ricordare che la seconda

trasmissione di direzione e velocità del vento avviene in modo corretto, ossia l'anemometro non inverte i dati. Superato tale test si ripete la medesima logica per il dato riguardante la velocità, per poi concludersi il test finale sul checksum che serve per assicurare integrità dei dati relativi esclusivamente alla direzione e velocità del vento della sola prima trasmissione, ossia quella che con i bit invertiti dal TX20. Superato quest'ultimo test, si ha la certezza che le informazioni sono integre e possono quindi venire visualizzate nei rispettivi campo "TextBlock" della UWP. Stampati i dati viene introdotto un ritardo di 1 secondo e viene riabilitata la chiamata al delegate col fine di processare nuovi datagrammi.

Segue il codice C# nel quale, oltre che a visualizzare i dati nella UWP, si è deciso anche di visualizzarli nella console, così da avere un debug in tempo reale anche qualora il Raspberry non sia collegato fisicamente ad un monitor. Per utilizzare il debug da console, come si vedrà in seguito, è necessario procedere all'esecuzione remota della UWP direttamente da Visual Studio e non dall'interfaccia web di Windows 10 IoT secondo gli esempi visti in precedenza.

```
public sealed partial class MainPage : Page
{
    private const int ritardoDatagrammaMicroSecondi = 1000000;
    private const int pinRicezioneTX20 = 4;
    private const int pinAttivazioneTrasmissioneTX20 = 17;
    private GpioController gpio;
    private GpioPin pinGPIO4;
    private GpioPin pinGPIO17;

    String[] elencoDirezioni = new String[16]
                                {
                                  "N","NNE","NE","ENE","E","ESE","SE","SSE",
                                  "S","SSW","SW","WSW","W","WNW","NW","NNW"
                                 };
```

```
private void letturaDatiDaTX20(GpioPin sender, GpioPinValueChangedEventArgs e)
{
  var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
  {
    if (e.Edge == GpioPinEdge.FallingEdge)
    {
       pinGPIO4.ValueChanged -= letturaDatiDaTX20;

       int bitsStartFrame = startFrame();
       int bitsDirezioneVento1 = direzioneVento1();
       int bitsVelocitaVento1 = velocitaVento1();
       int bitsChecksum = checksum();
       int bitsDirezioneVento2 = direzioneVento2();
       int bitsVelocitaVento2 = velocitaVento2();

       int calcoloChecksum = calcoloDelChecksumSulDatagramma
                                          (bitsDirezioneVento1, bitsVelocitaVento1);
       if (
              bitsStartFrame == 4 &&
              bitsDirezioneVento1 == bitsDirezioneVento2 &&
              bitsVelocitaVento1 == bitsVelocitaVento2 &&
              calcoloChecksum == bitsChecksum
         )
       {
          Debug.Write("------------------------------------------------------------\n");
          Debug.Write("La direzione del vento è = " +
                                    elencoDirezioni[bitsDirezioneVento1].ToString());
          Debug.WriteLine(" --- La velocità del vento è pari a {0} m/s ({1} km/h - {2}
                          mph)", bitsVelocitaVento1.ToString(),
                          ((float)bitsVelocitaVento1 * 3.6).ToString(),
                          ((float)bitsVelocitaVento1 * 2.2).ToString()
                       );
```

```
            Debug.WriteLine("-------------------------------------------------------------");
            Debug.WriteLine("/////////////////////////////////////////////////////////////");

            txtDirezioneVento.Text = "Direzione vento = " +
                                     elencoDirezioni[bitsDirezioneVento1].ToString();
            txtVelocitaMS.Text = bitsVelocitaVento1.ToString() + " m/s";
            txtVelocitaKMH.Text = ((float)bitsVelocitaVento1 * 3.6).ToString() + " km/h";
            txtVelocitaMPH.Text= ((float)bitsVelocitaVento1 * 2.2).ToString() + " mph";
          }

          delay(ritardoDatagrammaMicroSecondi);

          pinGPIO4.ValueChanged += letturaDatiDaTX20;
        }
      }
    );
  }
```

Tutta la logica implementativa del flow chart deve essere incorporata all'interno del dispatcher asincrono, così da potere interagire direttamente con le componenti visuali della UWP, come già discusso nel precedente capitolo. Il parametro "sender" identifica il possessore del delegate, che è la GPIO4, quindi tale parametro è di tipo "GpioPin". Il secondo parametro "e" identifica gli argomenti che in modo automatico vengono passati al delegate, ossia a tale metodo, quindi il tipo di "e" risulta essere "GpioPinValueChangedEventArgs". Il test sulla variazione della soglia di tensione viene fatto sfruttando il metodo "Edge" di "e", valore di tensione che viene passato, come detto in precedenza, in modo automatico al delegate. Per descrivere il passaggio da stato logico alto a basso, si utilizza il campo "FallingEdge", pari a 0, del tipo enumerativo "GpioPinEdge". Viene disabilitato il delegato tramite la chiamata "pinGPIO4.ValueChanged -= letturaDatiDaTX20;" utilizzando l'operatore "-=" che serve a rimuovere dalla lista di delegate un metodo. Essendo "letturaDatiDalTX20" l'unica funzione ad essere stata in precedenza inserita, il risultato è una lista vuota. Successivamente sono chiamati nell'ordine le funzioni che restituiscono il valore decimale corrispondente ai bit

letti dall'onda quadra, dati che vengono memorizzati nelle rispettive variabili locali. Il controllo condizionale, che nel flow chart della Fig.233 è diviso in quattro blocchi, qui è unico e riporta tutte le condizioni che contemporaneamente devono essere soddisfatte, motivo per cui vi è la presenza del AND logico tra le varie condizioni. Il superamento di tutte le condizioni si traduce nella stampa a video dei rispettivi valori, anche se prima si procedere a visualizzare alla console di debug le medesime informazioni tramite la chiamata "Debug.WriteLine()", molto utile qualora il device IoT non disponga di uno strumento di visualizzazione, come accade spesso per dispositivi quali le schede embedded. I punti cardinali da visualizzare sono 16 e sono memorizzati in un vettore di stringhe chiamato "elencoDirezioni[ ]" come dato membro di classe. Il campo che indica la direzione è lungo 4 bit, quindi le possibili combinazioni sono appunto 2^4=16. Quando la funzione "direzioneVento1()" restituisce il valore nel dato locale "bitsDirezioneVento1", questo viene usato come indice del vettore per indicare il punto cardinale corretto. Un valore pari a 0 rappresenta il nord, mentre un indice pari a 15 indica nord-nord-ovest. La Fig.234 riporta i punti cardinali con il rispettivo indice letto dalla funzione "direzioneVento1()" e "direzioneVento2()".

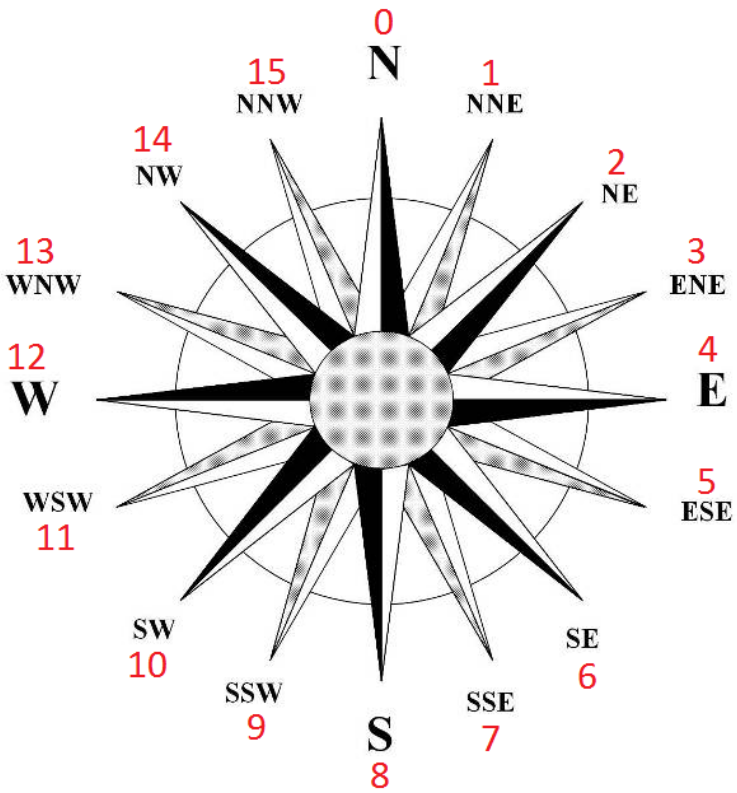


Fig.234

Il valore della velocità del vento viene rilevato in metri al secondo, quindi per la trasformazione in km/h e mph è necessario effettuare l'opportuna moltiplicazione per 3.6 e 2.2 rendendo però il dato di tipo reale, altrimenti con una variabile intera non è rappresentabile l'informazione ottenuta dalla moltiplicazione.

Una volta visualizzati i dati, non resta che introdurre un ritardo simbolico in attesa del datagramma successivo, per tale scopo si è scelto un ritardo di 1 secondo che però deve venire espresso in micro secondi, motivo per cui il dato membro di classe "ritardoDatagrammaInMicroSecondi" vale 10^6 micro secondi. Questo lasso di tempo deve essere pensato nell'ottica dei 2 secondi tra l'invio di un datagramma ed il successivo. Il passo finale consiste nel riattivare il meccanismo del delegate, così da iterare l'intera logica per i datagrammi successivi.

Per avviare il debug da console, premere il pulsante "Remote Machine" direttamente dall'ambiente Visual Studio come si vede in Fig.235.

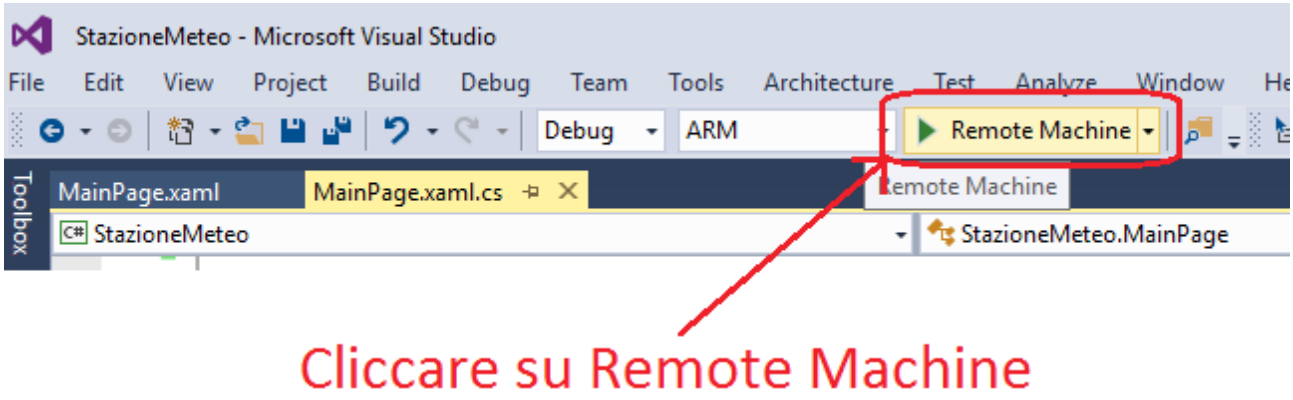


Fig.235

L'avvio in modalità Debug della UWP può impiegare qualche lasso di tempo, molto è legato alla dimensione del file e quindi alla quantità di codice scritto e componenti visuali utilizzate. Qualora per qualche istante nella barra dei titoli di Visual Studio compaia la voce "(Non risponde)", attendere senza panico il termine delle operazioni che l'IDE esegue in background, fino a che il software torni completamente operativo. Una volta che la UWP è in esecuzione, selezionare in fondo la voce "Output" come da Fig.236, così da osservare nell'apposita sezione l'invio dei dati letti dal TX20 e riportati in Visual Studio grazie all'istruzione "Debug.WriteLine()".

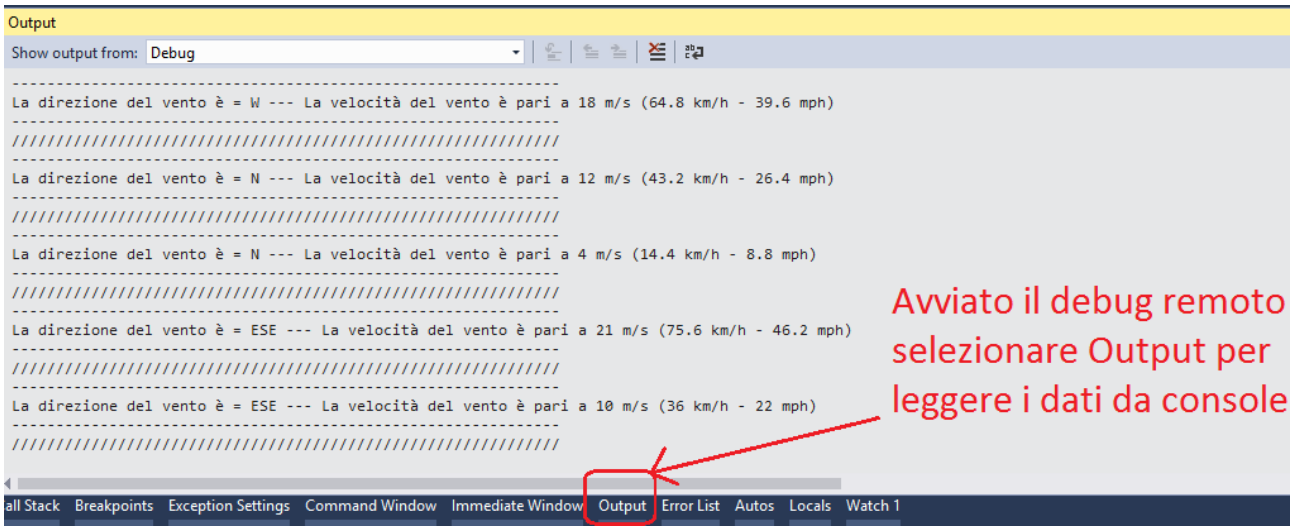


Fig.236

Ovviamente la UWP stampa i dati anche sul dispositivo di visualizzazione connesso al Pi, se presente. La Fig.237 mostra a run-time l'applicazione universale su un monitor 48".

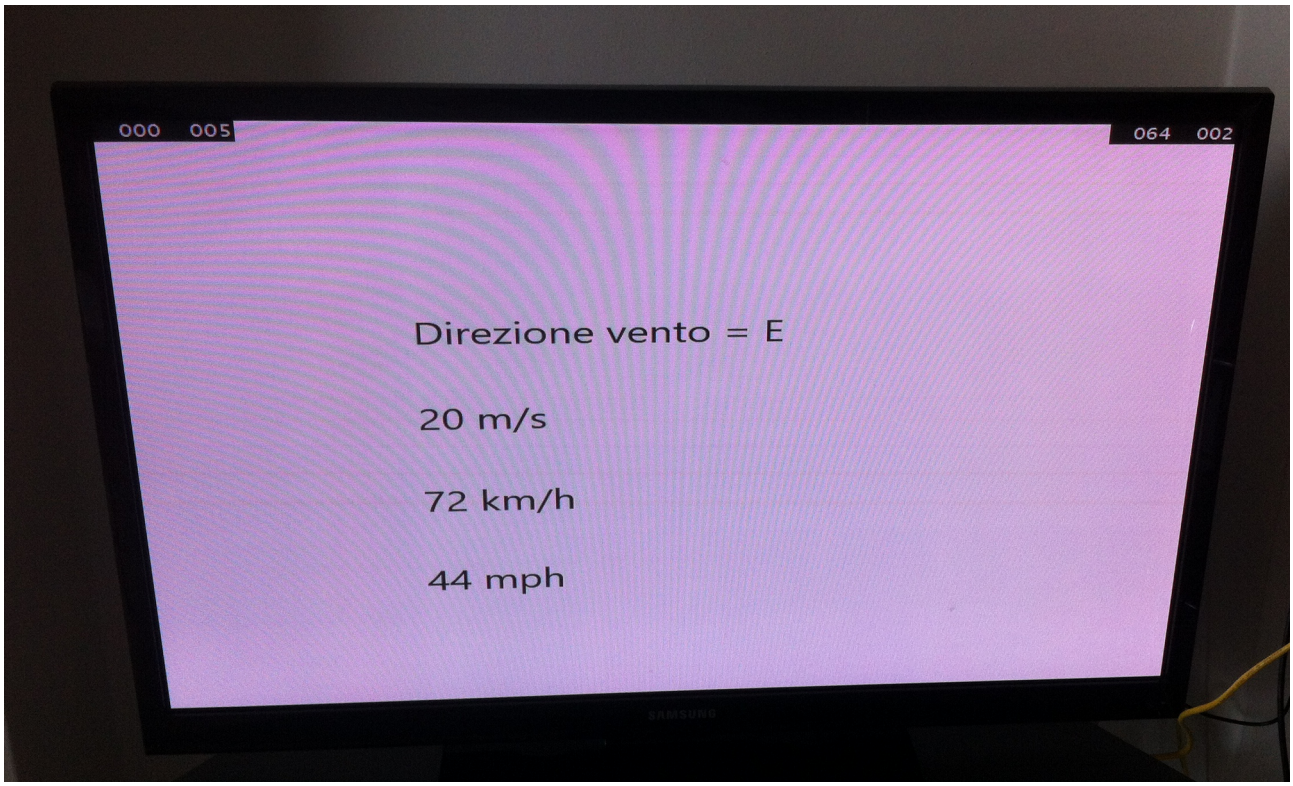


Fig.237

4.2.3.5 startFrame()

Questo metodo serve per estrapolare i primi 5 bit dello start frame analizzando la stato logico presente sul pin GPIO4. La Fig.238 riporta il diagramma di flusso del metodo.

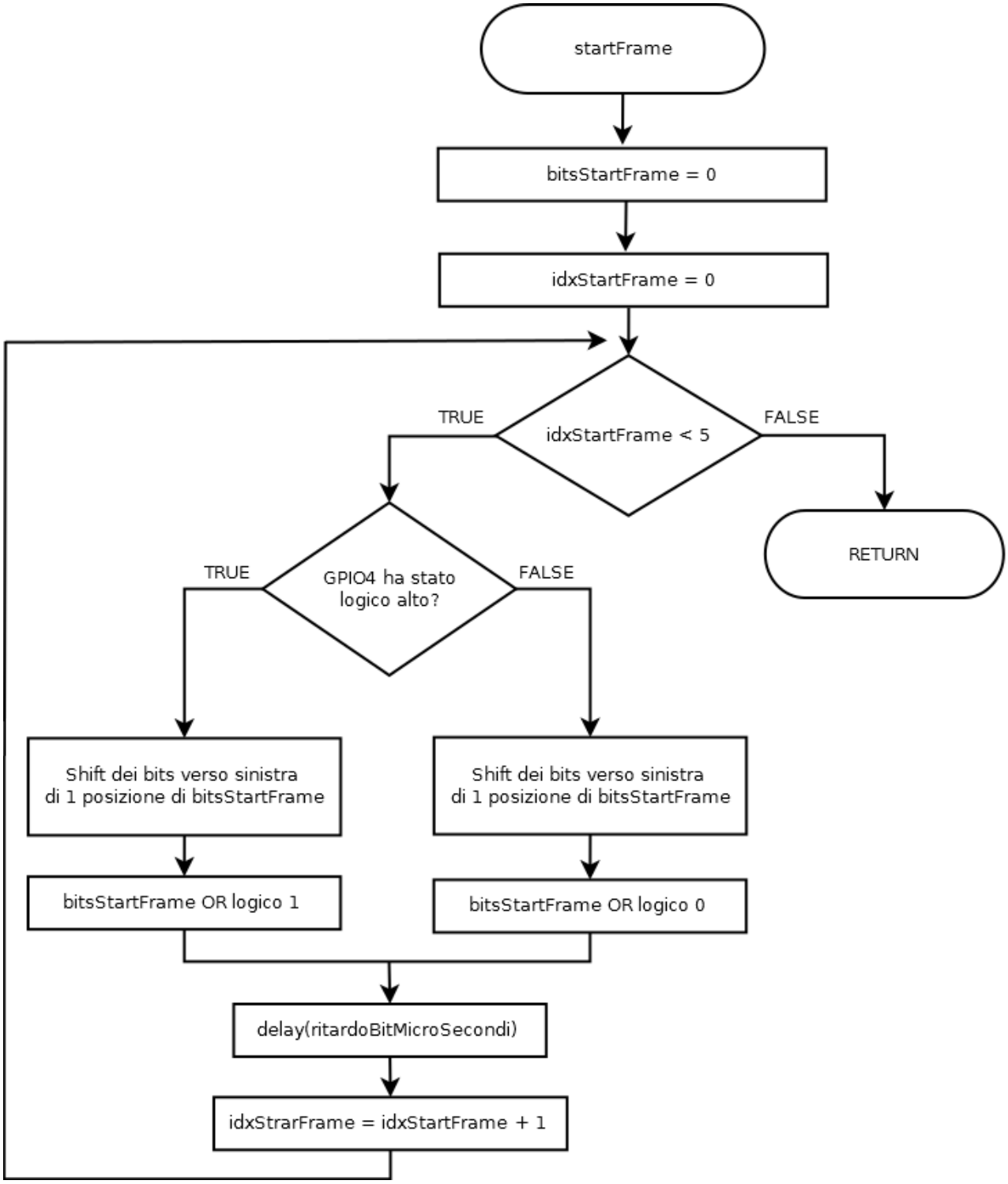


Fig.238

Per leggere i cinque bit si utilizza ovviamente un ciclo iterativo con indice iniziale posto a 0, valore molto comodo per la costruzione del dato intero "bitsDiStartFrame" che dovrà subire un meccanismo di bit wise tramite shiftamento dei bits. Questo dato locale viene inizializzato con 0.
Nel momento in cui la GPIO4 rileva tensione in ingresso, è necessario inserire un bit ad 1 nella posizione LSB, effettuando però prima uno shift dei bit verso sinistra così da preservare le letture precedenti. Il medesimo ragionamento vale se la GPIO4 rileva 0V, solo che il bit da inserire nella posizione LSB sarà ovviamente 0.
Tra una lettura e l'altra dei bit è necessario inserire un ritardo pari a 1220 micro secondi, ritardo memorizzato nel dato membro "ritardoBitMicroSecondi". Segue il codice C#.

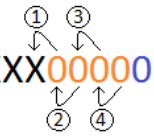
```
public sealed partial class MainPage : Page
{
    private const int ritardoBitMicroSecondi = 1220;
    private const int ritardoDatagrammaMicroSecondi = 1000000;
    private const int pinRicezioneTX20 = 4;
    private const int pinAttivazioneTrasmissioneTX20 = 17;
    private GpioController gpio;
    private GpioPin pinGPIO4;
    private GpioPin pinGPIO17;

    int startFrame()
    {
        int bitsDiStartFrame = 0;

        #if PRINT
            Debug.Write("Start frame = ");
        #endif

        for (int idxStartFrame = 0; idxStartFrame < 5; idxStartFrame++)
        {
            if (pinGPIO4.Read() == GpioPinValue.High)
            {
```

```
            #if PRINT
                Debug.Write("1");
            #endif

            bitsDiStartFrame = ((bitsDiStartFrame << 1) | 1);
        }
        else
        {
            #if PRINT
                Debug.Write("0");
            #endif

            bitsDiStartFrame = ((bitsDiStartFrame << 1) | 0);
        }

        delay(ritardoBitMicroSecondi);
    }

    return bitsDiStartFrame;
}
```

La parte di codice un pò ostica da capire è la manipolazione bit a bit del dato intero "bitsDiStartFrame". E' importante ricordare la regola protocollare dell'invio dello start frame. I bit originariamente vengono inviati invertiti, come sequenza di 11011. La parte di inversione hardware genera la sequenza corretta come 00100 che notiamo essere palindroma, osservazione molto importante perché la sequenza di trasmissione avviene sempre partendo dal bit LSB, e non dal MSB, quindi in fase di costruzione del dato è necessario gestire in modo opportuno quest'aspetto. Essendo la sequenza palindroma non è necessario, solo in questo caso, gestire la ricostruzione corretta dei bit LSB/MSB. Il dato su cui si lavora è la variabile locale "bitsDiStartFrame" che viene inizializzata a 0. La Fig.239 schematizza il modus operandi della manipolazione bit a bit.

Fase 0 (bitDiStartFrame=0) ---> 0000000000

1° BIT (arriva 1 logico) ---> XXXXX00000 shift verso sinistra XXXX00000



0000000000 OR 1

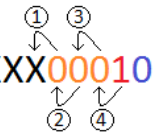
0000000000 OR
0000000001 =

0000000001

Fig.239

La fase 0 rappresenta l'inizializzazione del dato locale "bitsDiStartFrame" a 0 e per comodità sono stati indicati 10 bit di cui i cinque meno significativi colorati di arancione. Nel momento in cui arriva il primo 1 logico, si procede in primis a shiftare tutti i bit di una posizione verso sinistra, come si vede dalle frecce della figura, per poi effettuare a dato shiftato un OR logico con 1, memorizzando quindi il bit nella posizione LSB.
La Fig.240 mostra il secondo step.

2° BIT (arriva 1 logico) ---> XXXXX00001 shift verso sinistra XXXX00010



0000000010 OR 1

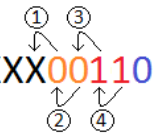
0000000010 OR
0000000001 =

0000000011

Fig.240

Ora il dato da shiftare è il risultato della prima fase, quindi XXXXX00001 al quale si applica un OR logico sempre con il bit ad 1.

3° BIT (arriva 0 logico) ---> XXXXX00011 shift verso sinistra XXXX00110

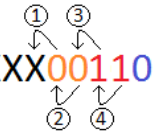



```
0000000110   OR   0

0000000110 OR
0000000000 =
----------
0000000110
```

Fig.241

Le fasi successive iterano il ragionamento, fino ad arrivare al dato finale della sequenza di start frame di Fig.243 che è XXXXX11011.

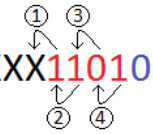
4° BIT (arriva 1 logico) ---> XXXXX00110 shift verso sinistra XXXX00110




```
0000001100   OR   1

0000001100 OR
0000000001 =
----------
0000001101
```

Fig.242

5° BIT (arriva 1 logico) ---> XXXXX01101 shift verso sinistra XXXX11010




```
0000011010   OR   1

0000011010 OR
0000000001 =
----------
0000011011
```

Fig.243

L'operatore logico che permette lo spostamento dei bit verso sinistra è "<<", così come il ">>" effettua lo shiftamento verso destra.
Nel codice C# l'operazione di shift anticipa quella di OR logico con il bit letto sulla porta GPIO4.

```
bitsDiStartFrame = ((bitsDiStartFrame << 1) | 1);
bitsDiStartFrame = ((bitsDiStartFrame << 1) | 0);
```

Scrivere bitsDiStartFrame << 1 comporta lo spostamento dei bit di 1 posizione verso sinistra, il risultato di questa operazione viene poi messo in OR con 1 oppure 0, sulla base del valore di tensione rilevato in ingresso sul pin GPIO4. Il risultato di queste due operazioni logiche viene poi memorizzato nel dato locale "bitsDiStartFrame" ed il ragionamento si itera tramite il ciclo for. Questa logica operativa è stata descritta in dettaglio perché viene utilizzata anche nei restanti metodi, seppure con qualche leggera variazione legata in primis alla trasmissione iniziale dei bit LSB.
Ogni bit della sequenza può essere debuggato in console tramite la chiamata "Debug.Write", la quale però è ora inserita in un blocco #if - #endif, come riportato qui sotto.

```
#define NOPRINT
using System;
...
...
#if PRINT
    Debug.Write("0");
#endif
```

L'istruzione di debug viene eseguita se e solo se C# definisce la macro PRINT tramite la direttiva #define in cima al codice, ma terminata la fase di testing conviene evitare la visualizzazione del singolo bit letto dall'onda per avere un output più pulito, quindi la macro PRINT viene modifica in NOPRINT.

4.2.3.6 direzioneVento1()

Questo metodo serve per estrapolare i successivi 4 bit che rappresentano la direzione del vento, bit che sono spediti dal TX20 invertiti e poi sistemati via hardware. Il diagramma di flusso di Fig.244 descrive la logica della funzione.

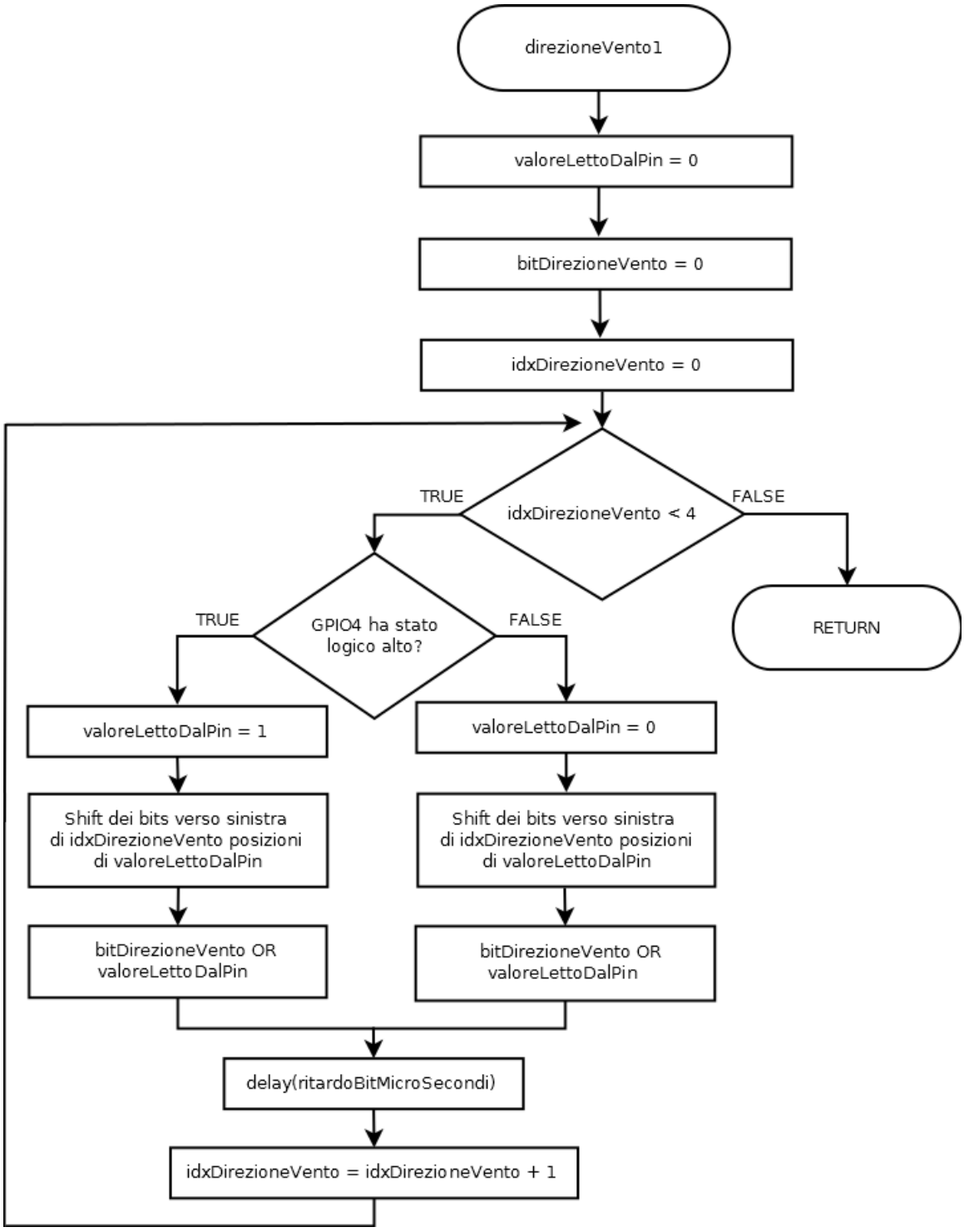


Fig.244

Il ragionamento da seguire è il medesimo analizzato nel paragrafo precedente, anche se ora è necessario gestire la trasmissione iniziale dei bit LSB e ricostruire la sequenza in modo corretto. La Fig.245 mostra un esempio di sequenza di bit.

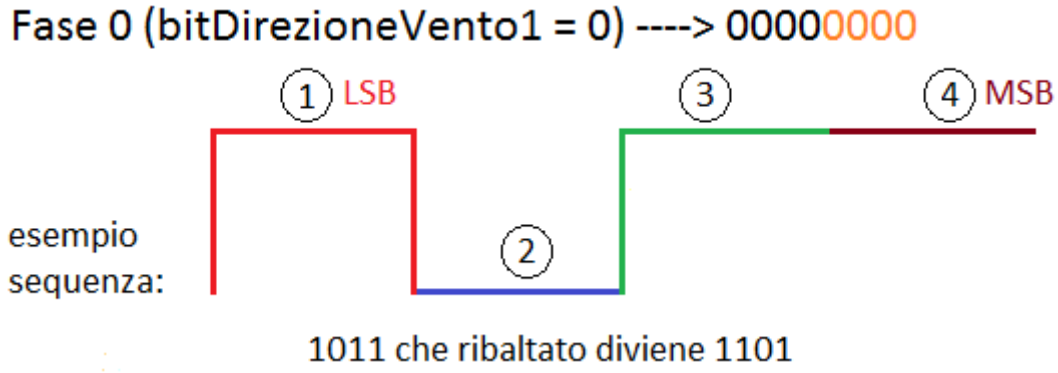


Fig.245

Il dato locale "bitDirezioneVento1" memorizza la sequenza di bit spedita dal TX20 e, grazie al meccanismo di bit wise, il corretto posizionamento dei bit avviene immediatamente. La Fig.246 mostra la ricezione del primo bit posto ad "1", il quale non subisce nessun meccanismo di shift verso sinistra, essendo questo il bit meno significativo. Il suo valore viene quindi posto in OR logico con il dato iniziale della variabile "bitDirezioneVento1" che è 0, quindi il risultato finale di questa prima fase è il numero 1.

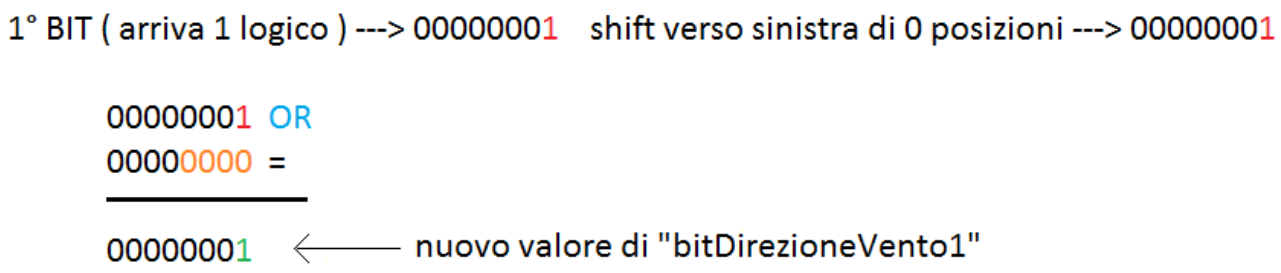


Fig.246

Successivamente arriva il secondo bit posto a "0" e nella sequenza di Fig.247 si lavora logicamente sul bit LSB indicato in rosso, facendolo shiftare di 1 posizione verso sinistra per poi mettere in OR logico il risultato dello scorrimento, con il risultato precedente di Fig.246 memorizzato nel dato locale "bitDirezioneVento1".

2° BIT (arriva 0 logico) ---> 00000000 shift verso sinistra di 1 posizione---> 00000000

```
00000000 OR
00000001 =
--------
00000001   <------  nuovo valore di "bitDirezioneVento1"
```

Fig.247

Il terzo bit posto ad "1" viene shiftato di 2 posizioni verso sinistra e messo in OR logico con il risultato finale di "bitDirezioneVento1" di Fig.247. In verde il bit LSB della fase iniziale, in rosso il terzo bit trasmesso. In Fig.248 è schematizzato questo passaggio.

3° BIT (arriva 1 logico) ---> 00000001 shift verso sinistra di 2 posizioni ---> 00000100

```
00000100 OR
00000001 =
--------
00000101   <------  nuovo valore di "bitDirezioneVento1"
```

Fig.248

Il quarto ed ultimo bit vale "1" e viene shiftato di 3 posizioni, così che inserito nella sequenza finale possa rappresentare il bit MSB complessivo. La Fig.249 riassume quest'ultima fase e, come si vede chiaramente, grazie agli shiftamenti incrementali verso sinistra il valore finale contenuto nel dato locale "bitDirezioneVento1" risulta correttamente invertito in termini di bit MSB/LSB.

4° BIT (arriva 1 logico) ---> 00000001 shift verso sinistra di 3 posizioni ---> 00001000

```
00001000 OR
00000101 =
--------
00001101   <------  valore finale di "bitDirezioneVento1"
```

Fig.249

Lo spostamento dei bit verso sinistra viene realizzato in C# tramite l'operatore logico "<<" secondo la sintassi "variabile << n_spostamenti_verso_sinistra", in questo modo è possibile poi effettuare un OR logico sfruttando la sintassi:

bitDirezioneVento1 = bitDirezioneVento1 | (valoreBitLettoDaOndaQuadra << posizione)

Per rendere più compatto questo codice è possibile utilizzare l'operatore "|=". Il valore della variabile "posizione" deve variare da 0 a 3, quindi è possibile utilizzare l'indice del ciclo iterativo per automatizzare lo scorrimento dei bit verso sinistra. Il codice C# che segue riassume quanto esposto fino ad ora.

```
int direzioneVento1()
    {
        int valoreLettoDalPin = 0;
        int bitDirezioneVento = 0;

        #if PRINT
            Debug.Write("\nDirezione vento n.1 = ");
        #endif

        for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
        {
            if (pinGPIO4.Read() == GpioPinValue.High)
            {
                valoreLettoDalPin = 1;

                bitDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

                #if PRINT
                    Debug.Write("1");
                #endif
            }
            else
```

```
        {
            valoreLettoDalPin = 0;

            bitDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

            #if PRINT
                Debug.Write("0");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    return bitDirezioneVento;
}
```

Sempre per motivi di testing è possibile abilitare la macro PRINT per stampare la sequenza dei singoli bit della direzione del vento ma, come già accennato in precedenza, si rischia di avere un output di console molto confuso, quindi si suggerisce la definizione della macro NOPRINT secondo quanto esposto in precedenza.
Ovviamente la scelta del bit a "1" o "0" dipende dallo stato logico letto dalla GPIO4, quindi si procede ad inizializzare in modo corretto il dato locale "valoreLettoDalPin" i cui bit verranno poi debitamente shiftati verso sinistra, secondo quando esposto in tale paragrafo, di "idxDirezioneVento" posizioni, ossia il valore dell'indice del ciclo iterativo for.

4.2.3.7 velocitaVento1()

Questo metodo serve per estrapolare i successivi 12 bit che rappresentano la velocità del vento, bit che sono spediti dal TX20 invertiti e poi sistemati via hardware. Il diagramma di flusso di Fig.250 descrive la logica della funzione.

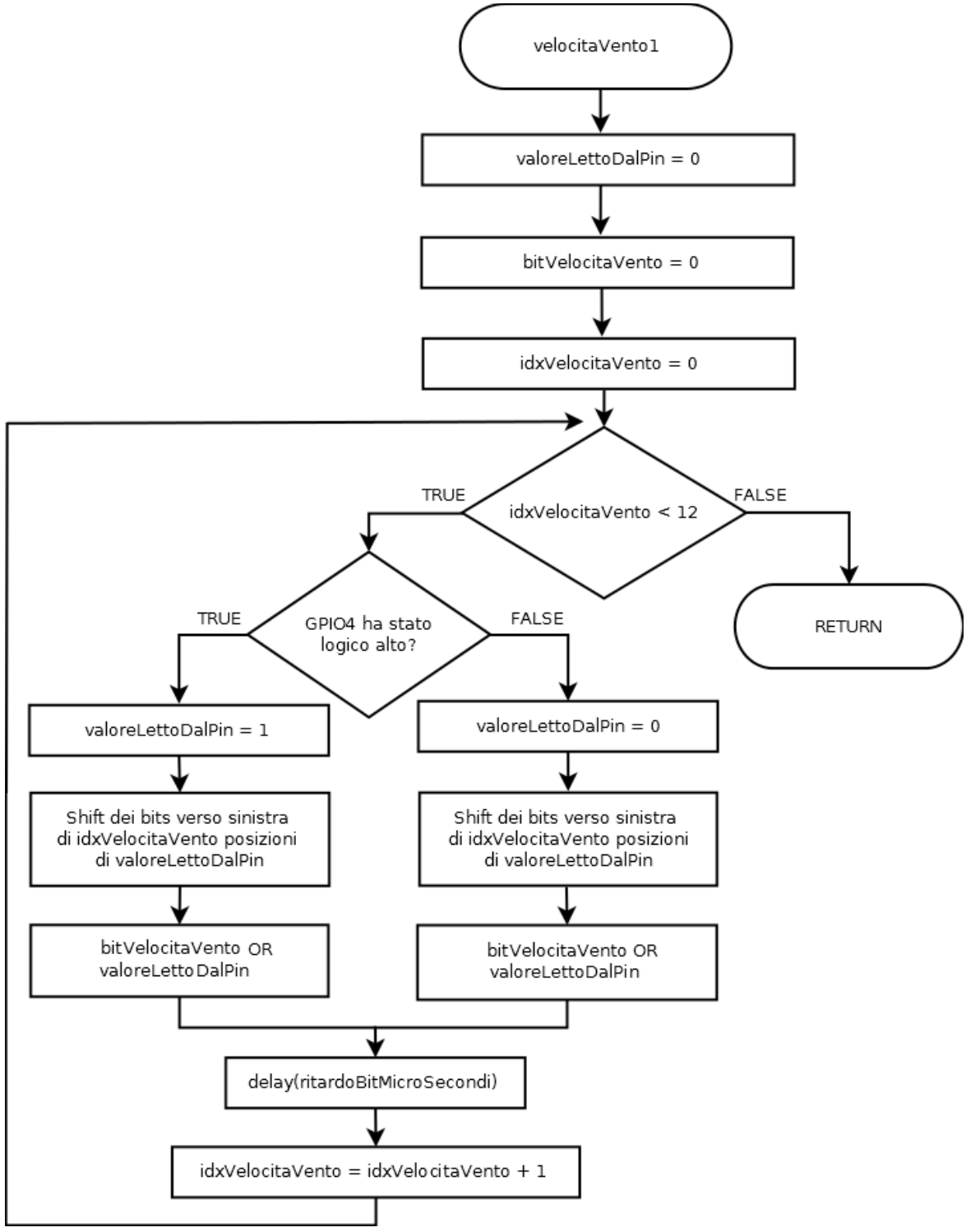


Fig.250

La logica per manipolare e gestire in modo corretto la sequenza dei 12 bit è la stessa adottata per il calcolo della direzione del vento, con l'unica variazione inerente le 12 iterazioni del ciclo for. Per quel che riguarda l'uso dell'indice "idxVelocitaVento", come indice di spostamento dei bit verso sinistra, vale quanto esposto in precedenza. Segue il codice C#.

```
int velocitaVento1()
{
    int valoreLettoDalPin = 0;
    int bitVelocitaVento = 0;

    #if PRINT
        Debug.Write("\nVelocità vento n.1 = ");
    #endif

    for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 1;

            bitVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                Debug.Write("1");
            #endif
        }
        else
        {
            valoreLettoDalPin = 0;

            bitVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);
```

```
            #if PRINT
                Debug.Write("0");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    bitVelocitaVento &= 511;

    return bitVelocitaVento;
}
```

E' importante ricordare che dei 12 bit che rappresentano la velocità del vento, solo i 9 bit meno significativi contengono l'informazione, infatti i primi 3 bit MSB sono sempre posti a 0. La massima velocità esprimibile risulta quindi essere 2^9-1=511 ossia 51.1 m/s. Per scrupolo nel codice C#, prima di restituire il dato al chiamante, viene fatta una maschera tramite un AND logico proprio con 511, così da assicurarsi che i bit, ad esclusione dei 9 LSB della variabile locale "bitVelocitaVento1", siano sempre nulli.

4.2.3.8 checksum()

Questo metodo serve per estrapolare i 4 bit del checksum che sono sempre spediti dal TX20 invertiti e poi sistemati via hardware. Il diagramma di flusso di Fig.251 descrive la logica della funzione.

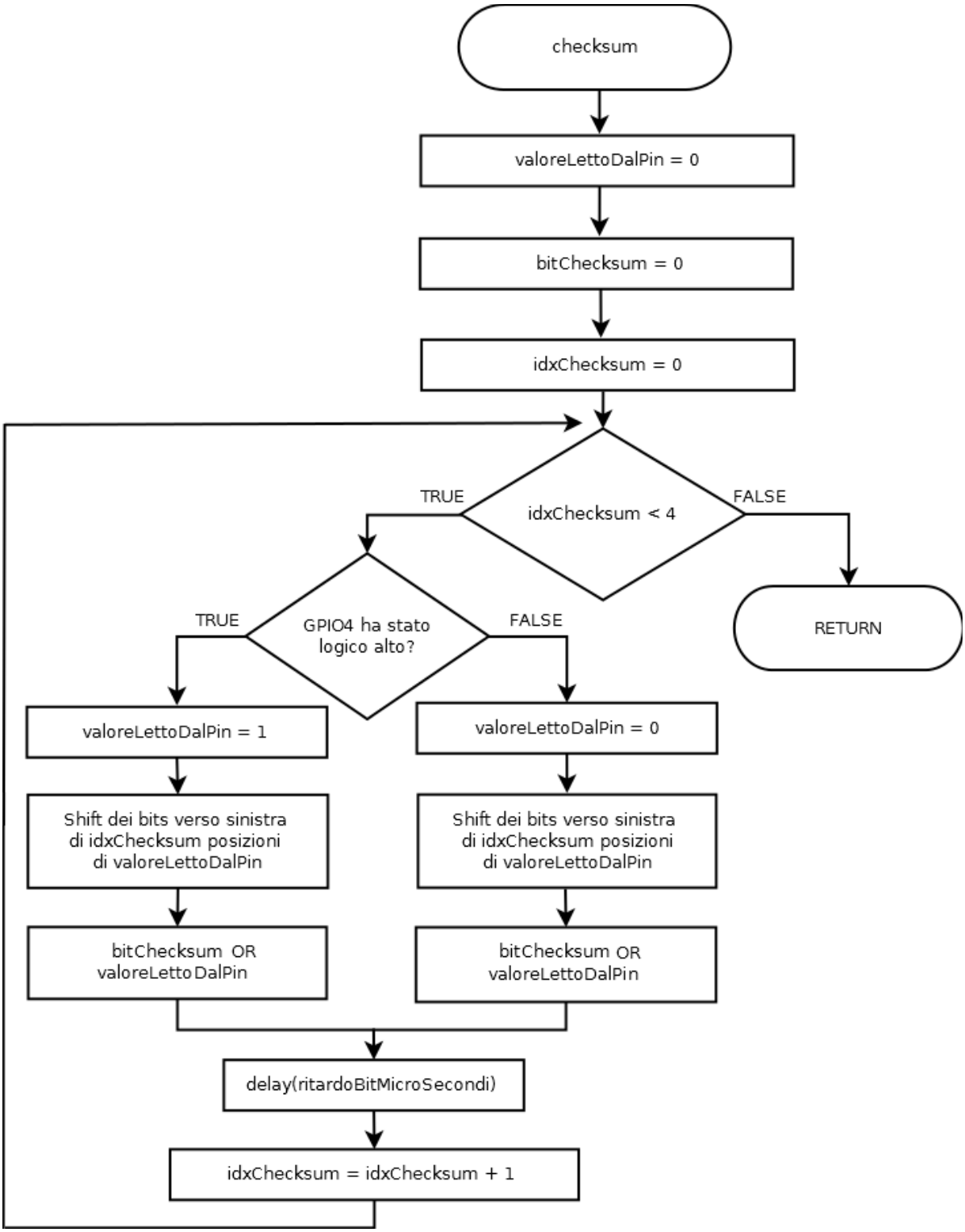


Fig.251

I bit della sequenza di checksum sono necessari per la verifica dell'integrità della prima sequenza relativa alla velocità e alla direzione del vento secondo quanto esposto in precedenza. La logica di adattamento dei bit LSB è identica ai casi precedenti, sia in termini di analisi del singolo bit che nell'uso del bit wise.

```
int checksum()
{
    int valoreLettoDalPin = 0;
    int valoreChecksum = 0;

    #if PRINT
        Debug.Write("\nChecksum = ");
    #endif

    for (int idxChecksum = 0; idxChecksum < 4; idxChecksum++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 1;

            valoreChecksum |= (valoreLettoDalPin << idxChecksum);

            #if PRINT
                Debug.Write("1");
            #endif
        }
        else
        {
            valoreLettoDalPin = 0;

            valoreChecksum |= (valoreLettoDalPin << idxChecksum);

            #if PRINT
```

```
            Debug.Write("0");
        #endif
      }

      delay(ritardoBitMicroSecondi);
    }

    return valoreChecksum;
}
```

4.2.3.9 direzioneVento2()

Questo metodo serve per estrapolare i 4 bit della direzione del vento seconda trasmissione. La Fig.252 riporta la logica della funzione che è simile a "direzioneVento1()".

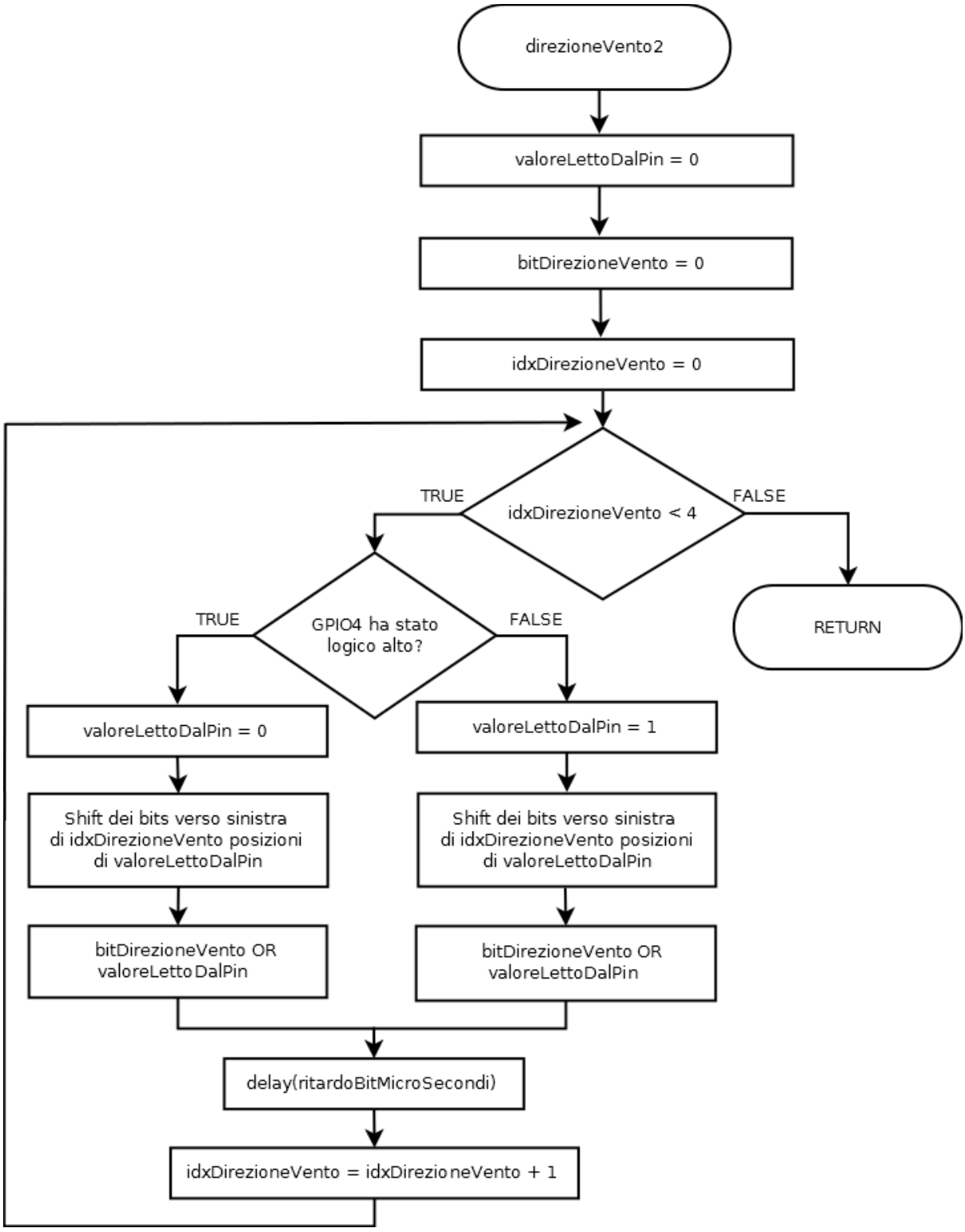


Fig.252

Rispetto alla prima trasmissione, i bit vengono spediti non invertiti da parte del TX20, quindi a livello software è necessario effettuare l'inversione logica. Osservando con attenzione il diagramma di flusso, si nota che dopo la verifica dello stato logico alto sul pin GPIO4, la variabile locale "valoreLettoDalPin" viene posta a "0", questo perché la circuiteria hardware ha in precedenza invertito il segnale spedito in modo corretto dall'anemometro. In maniera duale quando lo stato logico è basso, il dato locale "valoreLettoDalPin" viene posto ad "1" logico. Il resto del diagramma di flusso e del codice C# è praticamente identico al caso precedente, così come l'operazione di bit wise, varia solo l'assegnazione della variabile "valoreLettoDalPin" secondo quanto appena discusso.

```
int direzioneVento2()
{
        int valoreLettoDalPin = 0;
        int bitDirezioneVento = 0;

        #if PRINT
            Debug.Write("\nDirezione vento n.2 = ");
        #endif

        for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
        {
           if (pinGPIO4.Read() == GpioPinValue.High)
           {
              valoreLettoDalPin = 0;

              bitDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

              #if PRINT
                  Debug.Write("0");
              #endif
           }
           else
```

```
            {
                valoreLettoDalPin = 1;

                bitDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

                #if PRINT
                    Debug.Write("1");
                #endif
            }

            delay(ritardoBitMicroSecondi);
        }

        return bitDirezioneVento;
    }
```

4.2.3.10 velocitaVento2()

Questo metodo serve per estrapolare i 12 bit della direzione del vento seconda trasmissione. La Fig.253 riporta la logica della funzione che è simile a "velocitaVento1()".

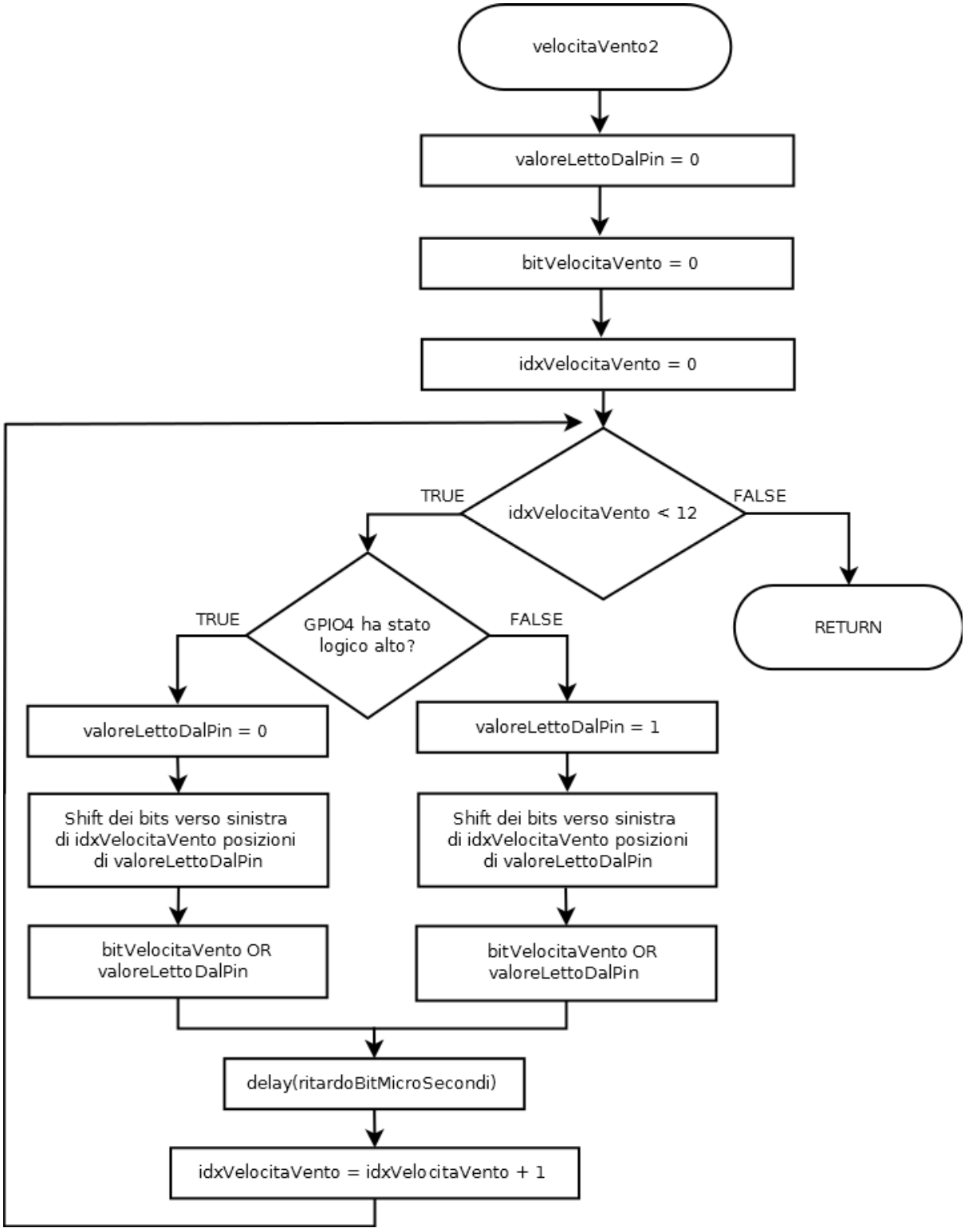


Fig.253

La velocità del vento seconda trasmissione viene spedita in modo corretto da parte del TX20, quindi va fatta la medesima riflessione della funzione "direzioneVento2()". La strategia risolutiva è identica, ossia l'inversione logica software lavorando direttamente sul valore assegnato al dato locale "valoreLettoDalPin". Il codice C# che segue è praticamente gemello del metodo "velocitaVento1()" con la sola variazione nelle due righe che assegnamento "0/1" alla variabile locale "valoreLettoDalPin".

```
int velocitaVento2()
{
        int valoreLettoDalPin = 0;
        int bitVelocitaVento = 0;

        #if PRINT
            Debug.Write("\nVelocità vento n.2 = ");
        #endif

        for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
        {
           if (pinGPIO4.Read() == GpioPinValue.High)
           {
              valoreLettoDalPin = 0;

              bitVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

              #if PRINT
                  Debug.Write("0");
              #endif
           }
           else
           {
              valoreLettoDalPin = 1;

              bitVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);
```

```
            #if PRINT
                Debug.Write("1");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    bitVelocitaVento &= 511;

    return bitVelocitaVento;
}
```

4.2.3.11 calcoloDelChecksumSulDatagramma()

Questo metodo calcola il checksum operando direttamente sulle informazioni, precedentemente acquisite, di direzione e velocità del vento. La Fig.254 mostra la logica implementativa della funzione.

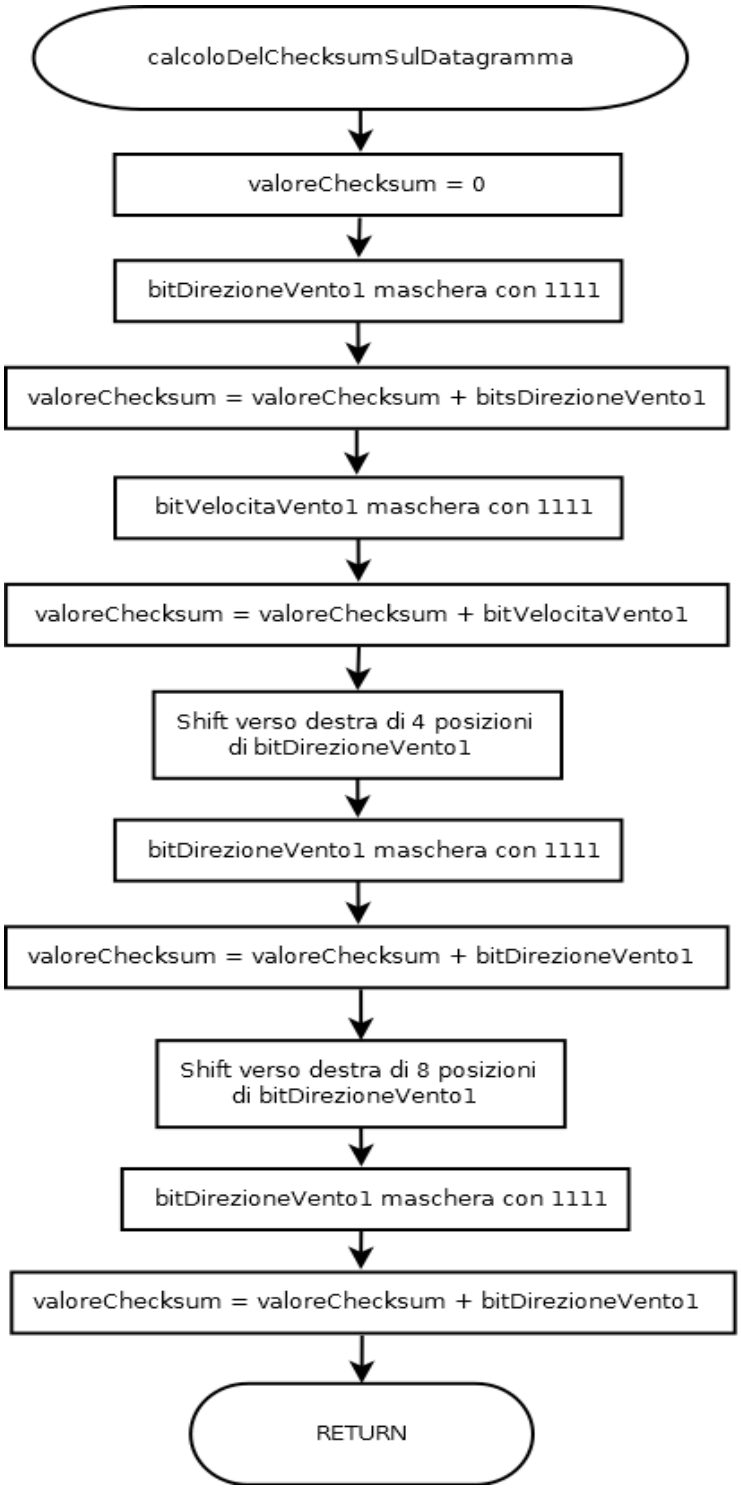


Fig.254

La funzione ha come parametri in ingresso la direzione e la velocità del vento inviati nella prima trasmissione, sui quali deve intervenire in modo da determinarne l'integrità. In sostanza è da farsi una somma logica tra i 4 bit della direzione del vento e tra le tre quaterne che compongono i 12 bit della velocità. Il risultato della somma andrà poi comparato con la sequenza dei 4 bit di checksum col fine di determinare la validità delle informazioni inviate nella prima trasmissione. Segue il codice C#.

```
int calcoloDelChecksumSulDatagramma(
                                        int bitsDirezioneVento1,
                                        int bitsVelocitaVento1
                                      )
{
    int valoreChecksum = 0;

    valoreChecksum += bitsDirezioneVento1 & 15;
    valoreChecksum += bitsVelocitaVento1 & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 4) & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 8) & 15;

    return valoreChecksum;
}
```

Il dato locale "valoreChecksum" viene inizializzato a "0" così che i bit siano correttamente resettati prima di effettuare l'operazione logica di somma. La prima somma che conviene effettuare, la più semplice, è tra il dato locale e la direzione del vento ricordando che le informazioni utili che sono presenti nel parametro in ingresso "bitsDirezioneVento1" sono esclusivamente i soli 4 bit meno significativi, ma essendo il dato di tipo "int", la sua lunghezza è pari a 64 bit, quindi conviene applicare una maschera ai soli 4 bit LSB. Per impostare la maschera che non alteri il contenuto di questi 4 bit, ma che assicuri a 0 tutti i restanti bit più significativi, conviene utilizzare l'operatore logico di AND con un numero binario del tipo "0.....01111", che tradotto in decimale equivale allo scalare "15". Il primo step della somma risulta quindi pari al valore della direzione del vento opportunamente mascherato e memorizzato nuovamente in "valoreChecksum". Gli

step successivi comportano di gestire, a fasi alterne, le tre quaterne che compongono i 12 bit della velocità del vento, partendo dai 4 bit LSB del parametro in ingresso "bitsVelocitaVento1". Seguendo la medesima strategia fatta per la direzione del vento, è conveniente mascherare i 4 bit LSB, tramite l'operatore di AND logico, usando lo scalare "15". Il risultato della maschera viene sommato al contenuto precedente, ed il risultato della somma è riportato nuovamente nel dato locale "valoreChecksum". Quanto descritto è indicato nella Fig.255 come step 1, nella quale la velocità del vento è rappresenta da una sequenza generica del tipo "XXXXYYYYZZZZ".

Fig.255

L'operazione di AND logico restituisce i bit meno significativi "ZZZZ", mentre i restanti sono a "0". La seconda fase deve effettuare la somma logica con i bit indicati come "YYYY", solo che questi sono ovviamente nella posizioni più avanzate rispetto ai 4 bit LSB. Il gioco consiste nello shiftare verso destra, "on the fly", tutti bit di 4 posizioni, così che la parte significativa "YYYY" si trovi nella posizione per subire il meccanismo il mascheramento sempre tramite lo scalare "15". Il risultato della maschera viene poi sommato al valore precedente. Nella terza ed ultima fase, la parte significativa "XXXX" deve venire shiftata di 8 posizioni, sempre "on the fly", così da posizionare la parte significativa al posto dei bit LSB col fine di applicare nuovamente la maschera con il valore "15". Il risultato finale in "valoreChecksum" è il checksum calcolato sui campi direzione e velocità del vento.

4.2.4 Il problema del ritardo

Il lettore probabilmente avrà notato che nel paragrafo precedente non si è parlato di ritardo, in particolare non si è analizzato il metodo "delay()", ultimo attore del programma C#. Il fine di questa funzione è di introdurre un lasso temporale, pari alla lunghezza del singolo bit di 1220µsec, necessario per leggere in modo corretto il singolo bit del datagramma inviato dal TX20. Lo stesso codice viene poi usato per introdurre un ritardo simbolico di 1 secondo tra la lettura di un datagramma ed il successivo. Prima di vedere il diagramma di flusso ed il codice C#, è necessario spendere due parole su come deve funzionare il ritardo o, in maniera duale, quale tipo di codice non si deve implementare a livello di .NET.
Esistono vari approcci per creare un ritardo in ambito C# .NET, sopratutto con le ultime release del framework, tra questi vale la pena di evidenziare:

- Thread.Sleep(<msec>)
- await Task.Delay(<msec>)
- Task.Delay(<msec>).Wait()

La chiamata "Thread.Sleep(<msec>)" è un modo classico per forzare il thread corrente ad un ritardo, peccato che il metodo statico "Sleep" non sia presente in .NET Core utilizzato per sviluppare applicazione UWP su Raspberry Pi 3, quindi in fase di scrittura non si potrà impiegare tale approccio.
La chiamata "await Task.Delay(<msec>)" viene utilizzata nella scrittura di codice non bloccante, quindi significa che dopo la sua esecuzione, il flusso di codice procede sequenzialmente all'istruzione successiva alla chiamata, quindi il suo impiego in una logica bloccante non porta nessun beneficio. Questo strumento è idoneo per porzioni di codice indipendente che deve venire eseguito in modo asincrono rispetto al flusso principale, porzione di codice che per qualche ragione necessità di subire un ritardo. La lettura dei bit dell'onda quadra inviata dal TX20 deve essere bloccante, ossia se non sono trascorsi i 1220µsec tra un bit ed il successivo, il codice deve restare in attesa che tale lasso di tempo sia trascorso. I primi due approcci sono quindi, sopratutto il secondo, fuori luogo. Il terzo metodo "Task.Delay(<msec>).Wait()" permette si di ottenere un ritardo bloccante, ma con scarsi risultati, visto che il problema che ora si presenta è quello

della precisione del ritardo, che risulta tutt'altro che pari a 1220µsec con la conseguente lettura errata di informazioni dall'anemometro.

4.2.4.1 Contatore hardware TSC

Per risolvere in modo accurato il problema dell'implementazione del ritardo, conviene appoggiarsi ad un contatore hardware presente all'interno del microprocessore chiamato TSC "Time Stamp Counter", anche se non tutte le CPU implementano tale dispositivo. Il processore ARM Cortex-A53 incorpora un TSC. Questo contatore lungo 64 bit, ha il semplice ruolo di temporizzatore e, ad ogni intervallo Δt, memorizza 1 tick sulla base della frequenza "f" dell'oscillatore al quarzo direttamente connesso al TSC. Un TSC non utilizza il clock della CPU, ma cosa più importante non deve mai variare il Δt di generazione del tick. In questa circostanza si parla di TSC invariante (o non variante) ed è il requisito essenziale per avere un ritardo che sia praticamente lo stesso nel tempo, salvo problemi di tolleranza legati al quarzo. Un altro aspetto importante è che il TSC deve evolvere nel tempo in modo continuo, memorizzando sempre tick ad intervalli Δt, senza subire influenze dal sistema operativo presente, come ad esempio stati di standby, hibernate o altro. Il TSC deve essere completamente indipendente sia da un punto di vista hardware che software. La Fig.256 riporta lo schema essenziale del TSC, nel quale esiste un generatore di temporizzazioni hardware, ossia l'oscillatore, che permette ogni Δt di tempo di incrementare, in modo indipendente, il valore contenuto nel TSC.

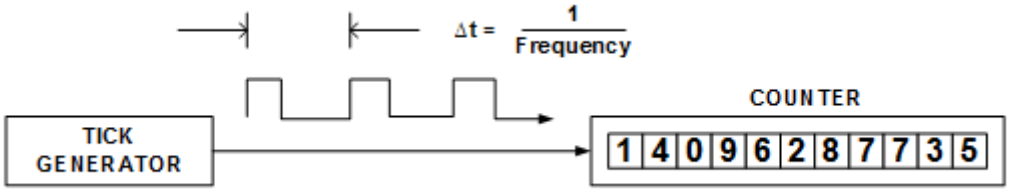


Fig.256

E' poi necessario, ovviamente, accedere alla lettura del contatore, azione che può non essere trascurabile. Il parametro della frequenza della figura soprastante è legato alle caratteristiche dell'oscillatore, semplice dispositivo al quarzo, o altro materiale, che permette di garantire una frequenza stabile ed accurata con una tolleranza espressa in "parts per milion", più comunemente indicata come "ppm".

4.2.4.2 Oscillatore

I personal computer moderni e i sistemi embedded montano oscillatori con tolleranze che vanno dai ±30ppm ai ±50ppm. Esistono processori con tolleranze più basse di quelle appena indicate, ma ovviamente i costi lievitano in modo netto, motivo per cui vengono sovente utilizzati in sistemi complessi come cluster o comunque soluzioni server business. Può essere interessante capire cosa significhi avere un'oscillatore con una tolleranza di ±50ppm.

Il lettore supponga di avere una CPU/Microcontrollore con una frequenza di 1MHz (1000000Hz), il cui oscillatore che fornisce il clock (quindi non si sta parlando in senso stretto del solo TSC), abbia una tolleranza di ±50ppm. La presenza di questa tolleranza fa si che il clock della CPU/Microcontrollore non sia 1MHz, ma bensì una frequenza variabile tra 999950Hz e 1000050Hz. Questa tolleranza di 50ppm, introduce di conseguenza un errore nella frequenza, non appunto si parla di errore di offset, che nel caso di ±50ppm è pari a ±0.000005. L'errore di offset è pari a 50 parti per milione ($50/10^6$) e comporta degli errori di misurazione nel tempo. E' interessante osservare il numero di secondi affetti da tale errore in una giornata. Un giorno possiede 24 ore, ossia 24*60=1440 minuti, quindi 1440*60=86400 secondi. Moltiplicando questo lasso di tempo in secondi per il valore di parti per milione, si ricava molto semplicemente i secondi di ritardo nella misurazione. Procedendo alla moltiplicazione ±(86400* 0.000005)=±4.3 secondi nell'arco di una giornata, con un'intervallo massimo di offset pari a 4.3*2=8.6 secondi.

Applicando tutti questi aspetti all'oscillatore del TSC, ne consegue che il conteggio dei tick subisce inevitabilmente un errore, il quale viene calibrato dai sistemi operativi moderni come Windows 8/10 o Linux.

L'oscillatore è quindi fondamentale per un corretto conteggio all'interno della CPU, ma senza ombra di dubbio non è la componente critica, visto che fattori più determinanti sono la risoluzione e la precisione del TSC. Il tempo è una componente continua, che evolve autonomamente e che non si può nè fermare nè riavvolgere. Rappresentare l'evoluzione del tempo significa utilizzare strumenti discreti, quindi entrare nel campo dei convertitori A/D e dell'ineliminabile errore di quantizzazione, che definisce la risoluzione del convertitore. L' errore di quantizzazione nel processo di conversione produce un'incertezza di misurazione del TSC di circa ± 1 tick, valore utilizzato per indicare la risoluzione del TSC stesso, indicata nei manuali come "tick resolution".

La Fig.257 mostra il gradino legato all'errore di quantizzazione nel caso si effettui una misurazione. In presenza di un gradino molto piccolo, l'errore di quantizzazione tra due misure può quasi essere considerato trascurabile.

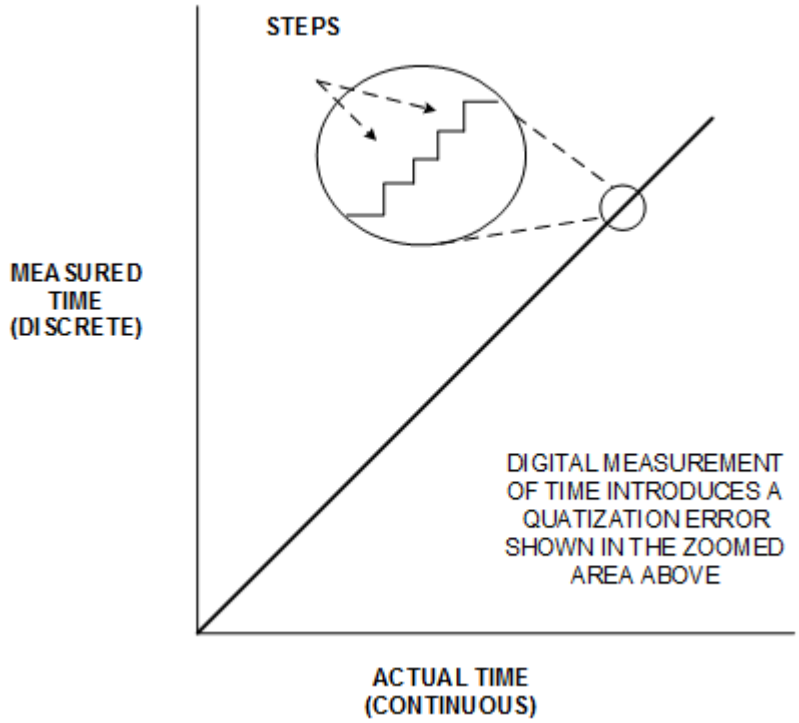


Fig.257

La risoluzione di un TSC è quindi un fattore importante, ma può rilevarsi del tutto insignificante in presenza di un altro fattore chiamato tempo di accesso. Quest'ultimo è un intervallo temporale, tra la richiesta di lettura del valore presente nel TSC, e l'effettiva restituzione del dato presente nel registro. La Fig.258 riporta un esempio di tempo di accesso.

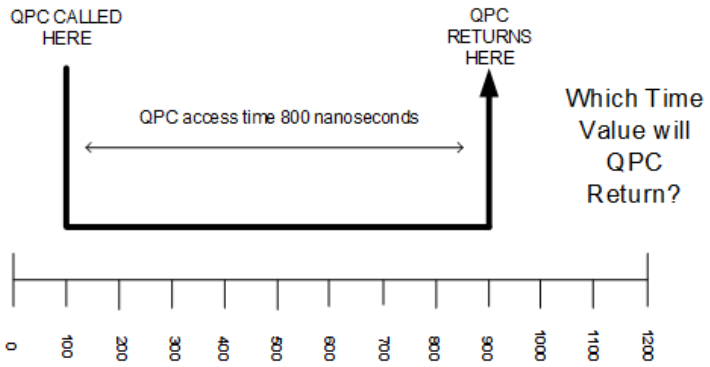


Fig.258

Come si nota nella scala graduata, gli intervalli sono di 100nsec, quindi tale valore rappresenta la risoluzione. L'accesso al TSC avviene dopo 100nsec, ma il dato viene restituito all'istante di tempo pari a 900nsec, che significa che sono trascorsi 800nsec prima di avere una risposta. In questo esempio la risoluzione di 100nsec è del tutto inutile, visto che qualsiasi accesso al TSC, da parte di un generico software, avrà sempre una precisione pari a 800nsec.
La Fig.259 mostra un altro esempio interessante, nel quale la risoluzione del TSC è sempre pari a 100nsec.

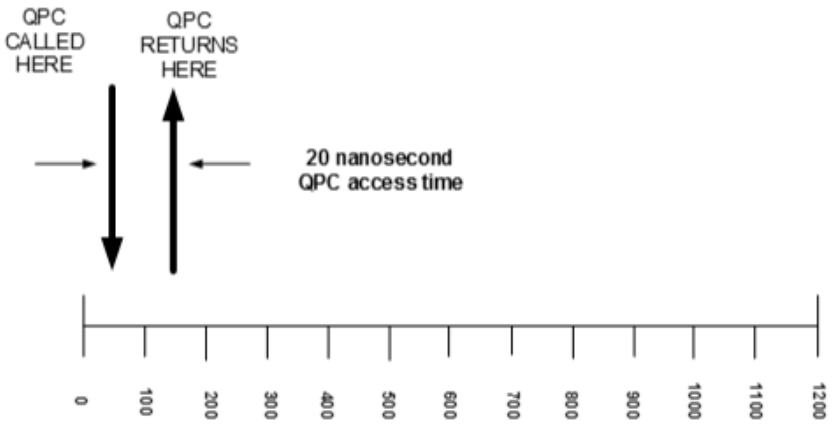


Fig.259

Questa volta l'accesso al TSC avviene ad un istante di tempo antecedente i 100nsec, mentre la restituzione del valore avviene subito dopo, con un intervallo di tempo complessivo pari a 20nsec, secondo quanto indicato in figura. Tale valore è nettamente più basso della risoluzione di 100nsec, quindi ne conviene che la precisione complessiva sarà, questa volta, legata alla risoluzione e non al tempo di accesso. Avere in questo esempio una risoluzione più bassa garantirebbe una precisione migliore lato software. La precisione è quindi il valore più grande tra risoluzione e tempo di accesso.
Per accedere al valore del TSC, Windows mette a disposizione delle API native chiamate Query Performance Counter o, più comunemente conosciute, QPC. Questo set di API sono state introdotte a partire dall'ormai obsoleto Windows 2000 e sono state migliorate negli anni per supportare i sistemi multi core e multi processore, nonché i sistemi di virtualizzazione ormai realtà dominante a livello globale. Esiste anche un set di API QPC dedicate alla modalità kernel, chiamate KeQPC, ma esulano da questa trattazione. Queste API sono un ottima strategia per impostare ritardi precisi nel tempo, sempre che il

contatore TSC sia invariante, altrimenti, come accennato in precedenza, si rischia di ottenere letture poco significative perché affette da standby o hibernate del sistema operativo. Le QPC hanno la caratteristica di lavorare esclusivamente con il TSC, non si appoggiano a nessun altro tipo di registro, quindi vi è la più totale indipendenza dall'UTC di sistema, così come non vi è la minima influenza da eventuali overclock automatici della frequenza di lavoro della CPU. E' possibile, ma sconsigliato, utilizzare delle istruzioni assembler per accedere al TSC, il set è conosciuto con il nome di RDTSC, anche se con l'avvento dei processori moderni il nome è stato cambiato in RDTSCP. La Microsoft stessa suggerisce di accedere, a basso livello, tramite le QPC. E' interessante capire se la CPU del calcolatore utilizzato ha o meno un TSC invariante, così come se supporta le nuove RDTSCP. Windows all'avvio effettua ovviamente tutti questi controlli. Per avere visione di queste informazioni, è necessario scaricare dal sito della Microsoft l'utility "Coreinfo v3.31", tramite il link https://technet.microsoft.com/it-it/sysinternals/cc835722 del technet. La Fig.260 riporta in parte la pagina.

Fig.260

Procedere al download cliccando su "Download Coreinfo" salvando il file compresso in una directory a piacere.

Dopo aver scompattato l'archivio compresso, aprire la shell dei comandi tramite la "cmd.exe" e, una volta entrati nel direttorio corretto, eseguire l'utility "coreinfo.exe" come da Fig.261. Viene riportato una parte dell'output, quello relativo alle caratteristiche del processore, al supporto delle istruzioni RDSTC e alla proprietà di invarianza del contatore TSC. E' lasciato al lettore la lettura degli altri parametri, tra cui le informazioni relative all'architettura a tre livelli della cache.

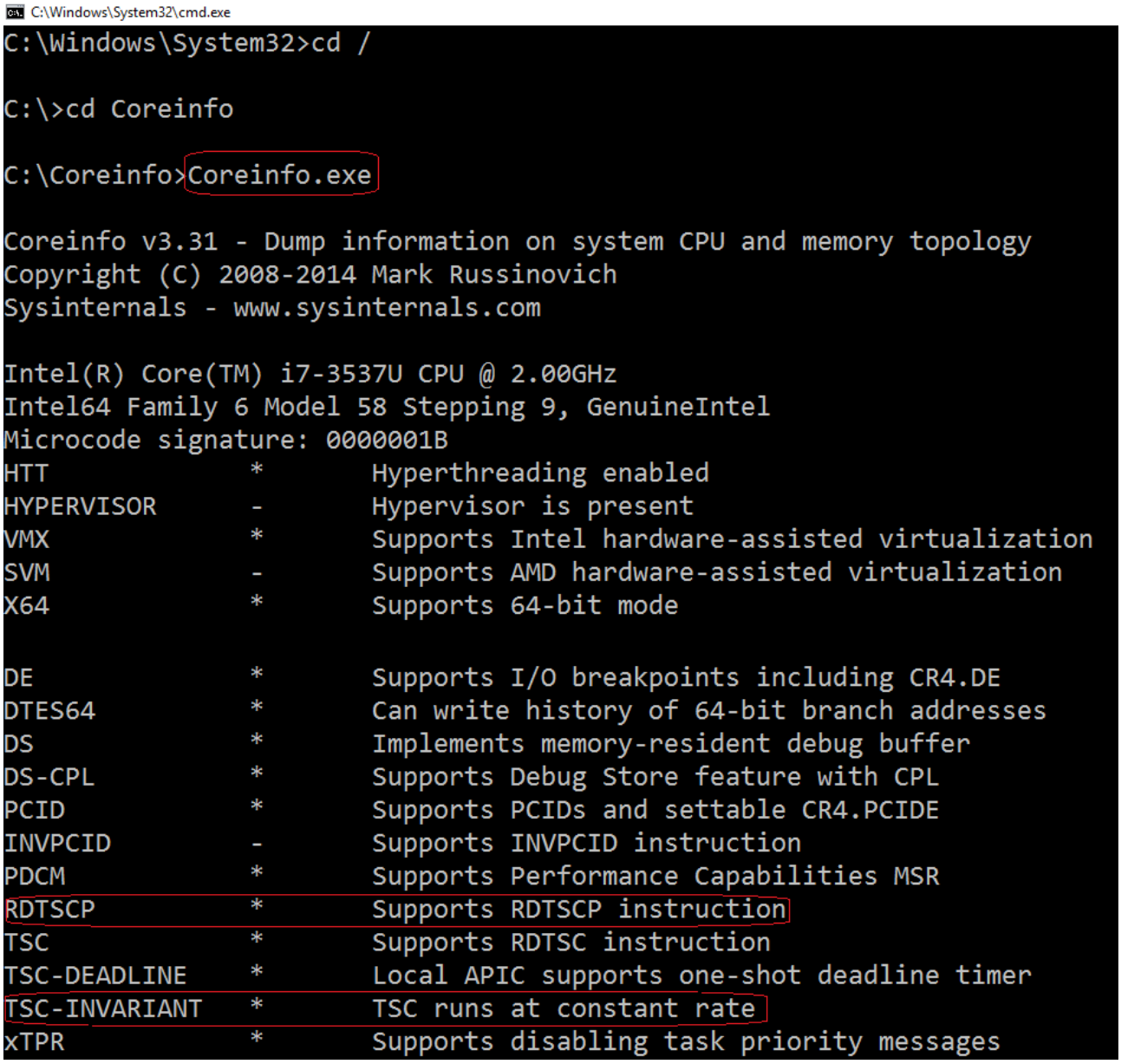


Fig.261

4.2.4.3 QPC ed implementazione della delay()

Tramite il linguaggio C/C++ è possibile richiamare direttamente le API QPC a livello utente, utilizzando le chiamate "QueryPerformanceCounter()" e "QueryPerformanceFrequency()". La prima API non fa altro che restituire il valore presente all'interno del TSC dal momento in cui la macchina è stata accesa, mentre la seconda API restituisce la frequenza con cui vengono generati i tick da parte dell'oscillatore. Segue un abbozzo di codice non gestito C/C++ che utilizza le suddette API.

```
LARGE_INTEGER tempoIniziale, tempoFinale, frequenzaDeiTick, intervalloTempo;

QueryPerformanceCounter(&TempoIniziale);
QueryPerformanceFrequency(&FrequenzaDeiTick);
QueryPerformanceCounter(&TempoFinale);

intervalloTempo.QuadPart = TempoFinale.QuadPart - TempoIniziale.QuadPart;
```

Queste poche righe di codice non fanno altro che calcolare il numero di tick che sono trascorsi tra le due chiamate di inizio e fine lettura. Nei sistemi a 32 bit la dimensione del LARGE_INTEGER utilizza due dati a 32 bit, mentre nei sistemi a 64 bit viene considerato come l'unione dei due dati, motivo per cui è necessario specificare l'opzione "QuadPart". L'esempio precedente è relativo di conseguenza ad una piattaforma 64 bit.
Qualora si voglia accedere al TSC in kernel mode, è possibile utilizzare la chiamata API "keQueryPerformanceCounter()".
In ambiente gestito C#/VB il framework .NET utilizza la classe "System.Diagnostics.Stopwatch" che permette di accedere al valore del TSC, classe che assieme ai relativi metodi viene utilizzata nel progetto corrente per realizzare il ritardo di precisione di 1220µsec e quello simbolico di 1 secondo.
Prima di vedere l'implementazione del codice C#, conviene analizzare il diagramma di flusso del metodo "delay()" di Fig.262 così da capire come realizzare un ritardo utilizzando il TSC. L'idea di base è sfruttare il valore presente nel contatore TSC in modo da calcolare il numero di tick presenti in un intervallo temporale di 1220µsec. Ovviamente il tick viene temporizzato dall'oscillatore, il quale impiegherà una data frequenza di temporizzazione.

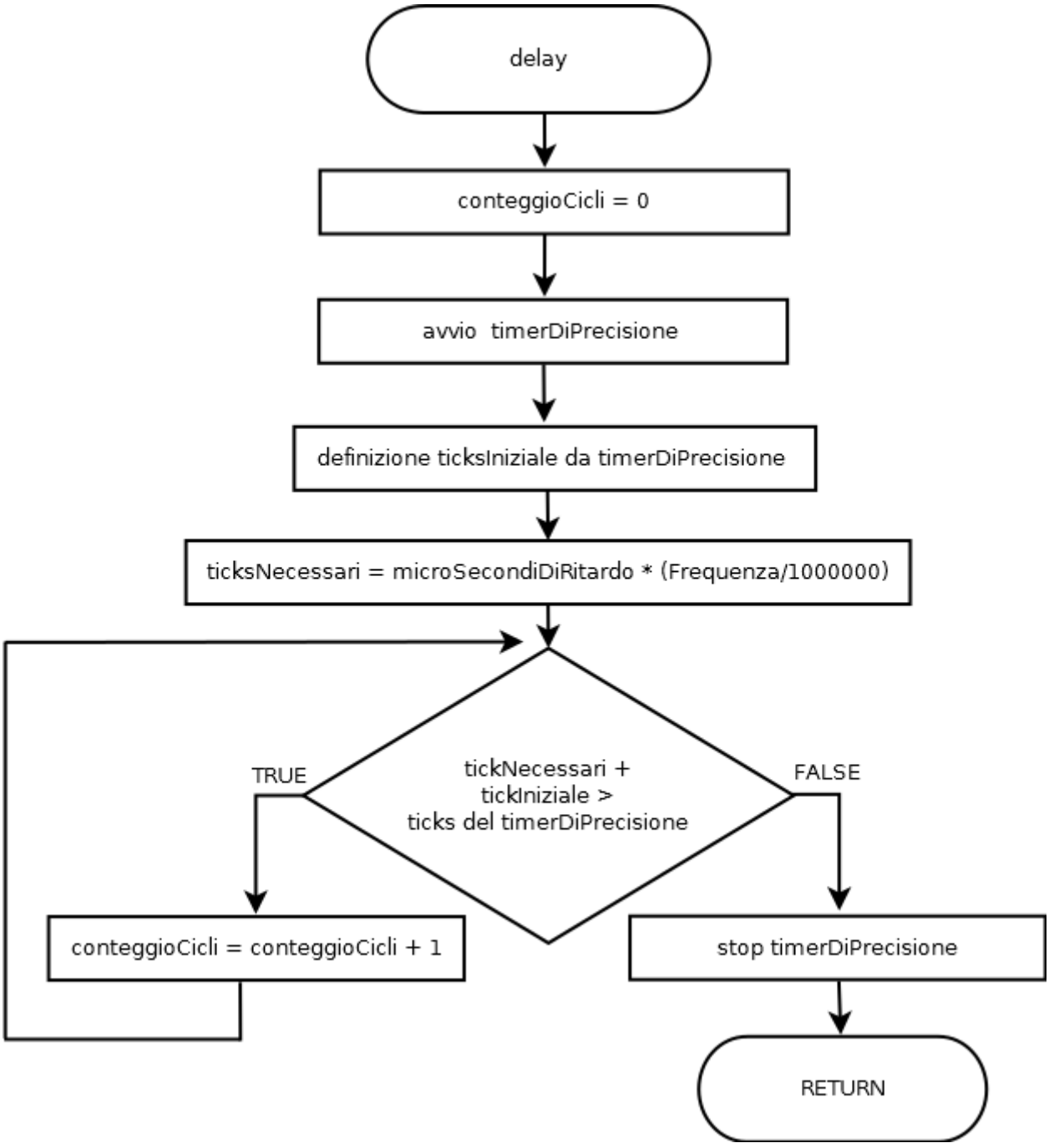


Fig.262

In sostanza va inizializzato l'oggetto timer di precisione, condizione necessaria per controllare il valore presente nel TSC in termini di tick conteggiati dall'avvio della macchina. Questo valore rappresenta il punto di partenza. Successivamente è necessario calcolare quanti tick sono presenti all'interno dell'intervallo di ritardo desiderato, calcolo che deve essere fatto sfruttando la frequenza di temporizzazione e garantendo, al tempo stesso, la scrittura di codice C# valido per processori ARM con caratteristiche diverse.

Nel momento in cui si carica la UWP su Raspberry Pi 2 o Pi 3 o evoluzioni future, il ritardo di 1220µsec deve essere eseguito in modo corretto, indipendentemente dal tipo di CPU e indipendentemente dal TSC che si spera, in versioni future, sia sempre invariante. La formula per il calcolo dei tick necessari è banale:

$$tick - necessari = \frac{Frequenza - TSC}{1.000.000} * "ritardo - voluto - microsec"$$

Il primo step è dividere la frequenza di temporizzazione per 1000000 visto che il ritardo che vogliamo eseguire viene indicato in micro secondi, questo ci permette quindi di conoscere il numero di tick presenti in 1µsec. Moltiplicando il valore della divisione per il ritardo voluto espresso in µsec, si ottiene il numero di tick complessivi che saranno presenti nell'intervallo di ritardo desiderato. Le informazioni ottenute fino a questo momento risultano essere il tick iniziale letto in precedenza e i tick necessari a rappresentare il ritardo voluto. Il tick finale può essere visto come la somma tra il tick iniziale e quelli necessari, come rappresenta indicativamente la Fig.263.

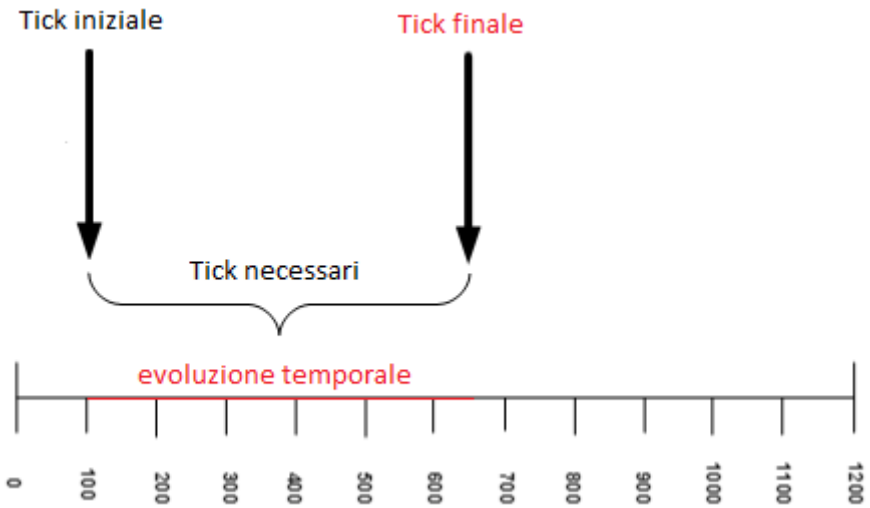


Fig.263

Finché non si raggiunge il tick finale, significa che il ritardo voluto non è trascorso, quindi si deve entrare in loop eseguendo un operazione generica come l'incremento di un contatore che indica il numero di cicli eseguiti. Il ritardo desiderato è considerato eseguito solo nel momento in cui si raggiunge o si supera il tick finale.

Il codice C# per la realizzazione del metodo "delay()" è molto semplice e sfrutta la classe "Stopwatch" del framework .NET Core. Il ritardo voluto viene passato come parametro in ingresso e corrisponde al numero di micro secondi desiderati. Per utilizzare il TSC è necessario inizializzare il timer tramite il metodo "StartNew" della classe "Stopwatch", leggendo poi dal TSC il valore dei tick correnti da quando il sistema operativo si è avviato, tramite il metodo "ElapsedTicks". Successivamente si procede al calcolo dei tick necessari, sfruttando la formula precedente, leggendo la frequenza di temporizzazione tramite il metodo "Frequency" della classe "Stopwatch" che restituisce un tipo di dato "long int". Per tipizzare nel modo corretto il risultato dell'operazione, viene convertito in double la frequenza, così come il ritardo voluto, così che il prodotto sia anch'esso double, per poi castizzare tutto in intero come numero di tick necessari. Successivamente basta implementare un semplice ciclo iterativo, nel quale la condizione legge il numero dei ticks trascorsi da quando il timer è stato inizializzato, tramite il metodo "StartNew", utilizzando il metodo "ElapsedTicks" e verifica se i tick trascorsi hanno raggiunto il numero di tick finali ad indicare che il ritardo è terminato. All'interno del ciclo si esegue un incremento di dato locale, come semplice operazione di attesa. Terminato il ritardo si blocca l'attività di misurazione dei tick trascorsi tramite il metodo "Stop". Segue il codice C#.

```
 void delay(int microSecondiDiRitardo)
{
   int conteggioCicli = 0;

   Stopwatch timerDiPrecisione = Stopwatch.StartNew();

   long ticksIniziale = timerDiPrecisione.ElapsedTicks;
   int ticksNecessari = (int)(
                              (double) microSecondiDiRitardo *
                              ((double) Stopwatch.Frequency / 1000000.0)
                            );

   while (timerDiPrecisione.ElapsedTicks < (ticksIniziale + ticksNecessari))
   {
      conteggioCicli++;
```

```
        }

        timerDiPrecisione.Stop();
    }
```

E' importante ribadire che altre implementazioni di ritardo che sfruttino il UTC "Coordinated Universal Time" non sono assolutamente adatti per realizzare ritardi di precisione, sopratutto nell'ambito di sistemi gestiti come C# .NET. L'impiego in C# della Task.Delay() sia sincrona che asincrona sfrutta il UTC, quindi è da evitare per tali tipi di applicazioni il suo impiego. Nel momento in cui il meccanismo di ritardo è non bloccante, la logica del codice è destinata a fallire, visto che il normale flusso di esecuzione del codice continua a venire eseguito. Per completezza di dettaglio sono riportate delle implementazioni di ritardi in C# non idonei a questo tipo di progetto.

```
    // tutti questi tipi di ritardi sfruttano UTC

    private void delay(double delay)
    {
      new ManualResetEventSlim(false).Wait(TimeSpan.FromMilliseconds(delay));
    }

    private void delay(double delay)
    {
        using (EventWaitHandle tmpEvent = new ManualResetEvent(false))
        {
            tmpEvent.WaitOne(TimeSpan.FromMilliseconds(delay));
        }
    }

    private void delay(double delay)
    {
      Task.Delay(-1).Wait(delay);
    }
```

4.2.5 Progetto UWP C#

Il capitolo ha descritto tutte le parti salienti di interfacciamento al Pi fino alla progettazione della scheda tramite AutoDesk Eagle. E' stato spiegato il modus operandi del TX20 e come realizzare, tramite C#, del codice gestito in UWP per la lettura dei dati. Le informazioni vengono quindi visualizzate nella semplice GUI, creata tramite i componenti messi a disposizione da Visual Studio e nella console di debug qualora il Pi non disponesse di uno strumento di visualizzazione, come in effetti accade quando viene posizionato all'esterno. In questo paragrafo si riporta per completezza tutto il codice del programma descritto nelle sezioni precedenti.

```
#define NOPRINT
using System;
using Windows.UI.Xaml.Controls;
using Windows.Devices.Gpio;
using Windows.UI.Core;
using System.Diagnostics;

namespace StazioneMeteo
{
   public sealed partial class MainPage : Page
   {
      private const int ritardoBitMicroSecondi = 1220;
      private const int ritardoDatagrammaMicroSecondi = 1000000;
      private const int pinRicezioneTX20 = 4;
      private const int pinAttivazioneTrasmissioneTX20 = 17;
      private GpioController gpio;
      private GpioPin pinGPIO4;
      private GpioPin pinGPIO17;

      String[] elencoDirezioni = new String[16] {
                                    "N","NNE","NE","ENE","E","ESE","SE","SSE",
                                    "S","SSW","SW","WSW","W","WNW","NW","NNW"
                                   };
```

```
public MainPage()
{
    this.InitializeComponent();

    attivazioneTX20();
}

private void attivazioneTX20()
{
    if (inizializzazioneGPIO() == false)
    {
        txtDirezioneVento.Text = "Errore inizializzazione GPIO!";

        return;
    }

    pinGPIO17 = gpio.OpenPin(pinAttivazioneTrasmissioneTX20);
    pinGPIO17.SetDriveMode(GpioPinDriveMode.Output);
    pinGPIO17.Write(GpioPinValue.High);
}

private void letturaDatiDaTX20(GpioPin sender, GpioPinValueChangedEventArgs e)
{
    var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        if (e.Edge == GpioPinEdge.FallingEdge)
        {
            pinGPIO4.ValueChanged -= letturaDatiDaTX20;

            int bitsStartFrame = startFrame();
            int bitsDirezioneVento1 = direzioneVento1();
            int bitsVelocitaVento1 = velocitaVento1();
            int bitsChecksum = checksum();
```

```
            int bitsDirezioneVento2 = direzioneVento2();
            int bitsVelocitaVento2 = velocitaVento2();

            int calcoloChecksum =
calcoloDelChecksumSulDatagramma(bitsDirezioneVento1, bitsVelocitaVento1);

            if (
               bitsStartFrame == 4 && bitsDirezioneVento1 == bitsDirezioneVento2 &&
               bitsVelocitaVento1 == bitsVelocitaVento2 && calcoloChecksum ==
bitsChecksum
             )
            {
               Debug.Write("-------------------------------------------------------------\n");
               Debug.Write("La direzione del vento è = " +
elencoDirezioni[bitsDirezioneVento1].ToString());
               Debug.WriteLine(" --- La velocità del vento è pari a {0} m/s ({1} km/h - {2}
 mph)", bitsVelocitaVento1.ToString(), ((float)bitsVelocitaVento1 * 3.6).ToString(),
((float)bitsVelocitaVento1 * 2.2).ToString());

               Debug.WriteLine("-------------------------------------------------------------");
               Debug.WriteLine("/////////////////////////////////////////////////////////////");

               txtDirezioneVento.Text = "Direzione vento = " +
                                        elencoDirezioni[bitsDirezioneVento1].ToString();
               txtVelocitaMS.Text = bitsVelocitaVento1.ToString() + " m/s";
               txtVelocitaKMH.Text = ((float)bitsVelocitaVento1 * 3.6).ToString() + " km/h";
               txtVelocitaMPH.Text = ((float)bitsVelocitaVento1 * 2.2).ToString() + " mph";
            }

            delay(ritardoDatagrammaMicroSecondi);

            pinGPIO4.ValueChanged += letturaDatiDaTX20;
         }
```

```
        }
        );
    }

    private bool inizializzazioneGPIO()
    {
        this.gpio = GpioController.GetDefault();

        if (gpio == null)
            return false;
        else
        {
            pinGPIO4 = gpio.OpenPin(pinRicezioneTX20);

            pinGPIO4.SetDriveMode(GpioPinDriveMode.Input);

            pinGPIO4.ValueChanged += letturaDatiDaTX20;

            return true;
        }
    }

    int startFrame()
    {
        int bitsDiStartFrame = 0;

        #if PRINT
            Debug.Write("Start frame = ");
        #endif

        for (int idxStartFrame = 0; idxStartFrame < 5; idxStartFrame++)
        {
            if (pinGPIO4.Read() == GpioPinValue.High)
```

```
      {
         #if PRINT
            Debug.Write("1");
         #endif

         bitsDiStartFrame = ((bitsDiStartFrame << 1) | 1);
      }
      else
      {
         #if PRINT
            Debug.Write("0");
         #endif

         bitsDiStartFrame = ((bitsDiStartFrame << 1) | 0);
      }

      delay(ritardoBitMicroSecondi);
   }

   return bitsDiStartFrame;
}

int direzioneVento1()
{
   int valoreLettoDalPin = 0;
   int valoreDirezioneVento = 0;

   #if PRINT
      Debug.Write("\nDirezione vento n.1 = ");
   #endif

   for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
   {
```

```
            if (pinGPIO4.Read() == GpioPinValue.High)
            {
                valoreLettoDalPin = 1;

                valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

                #if PRINT
                    Debug.Write("1");
                #endif
            }
            else
            {
                valoreLettoDalPin = 0;

                valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

                #if PRINT
                    Debug.Write("0");
                #endif
            }

            delay(ritardoBitMicroSecondi);
        }

        return valoreDirezioneVento;
    }

    int direzioneVento2()
    {
        int valoreLettoDalPin = 0;
        int valoreDirezioneVento = 0;

        #if PRINT
```

```
        Debug.Write("\nDirezione vento n.2 = ");
    #endif

    for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 0;

            valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

            #if PRINT
                Debug.Write("0");
            #endif
        }
        else
        {
            valoreLettoDalPin = 1;

            valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

            #if PRINT
                Debug.Write("1");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    return valoreDirezioneVento;
}
```

```
int velocitaVento1()
{
    int valoreLettoDalPin = 0;
    int valoreVelocitaVento = 0;

    #if PRINT
        Debug.Write("\nVelocità vento n.1 = ");
    #endif

    for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 1;

            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                Debug.Write("1");
            #endif
        }
        else
        {
            valoreLettoDalPin = 0;

            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                Debug.Write("0");
            #endif
        }

        delay(ritardoBitMicroSecondi);
```

```
    }

    valoreVelocitaVento &= 511;

    return valoreVelocitaVento;
}

int velocitaVento2()
{
    int valoreLettoDalPin = 0;
    int valoreVelocitaVento = 0;

    #if PRINT
        Debug.Write("\nVelocità vento n.2 = ");
    #endif

    for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 0;

            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                Debug.Write("0");
            #endif
        }
        else
        {
            valoreLettoDalPin = 1;

            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);
```

```
            #if PRINT
                Debug.Write("1");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    #if PRINT
        Debug.Write("\n");
    #endif

    valoreVelocitaVento &= 511;

    return valoreVelocitaVento;
}

int checksum()
{
    int valoreLettoDalPin = 0;
    int valoreChecksum = 0;

    #if PRINT
        Debug.Write("\nChecksum = ");
    #endif

    for (int idxChecksum = 0; idxChecksum < 4; idxChecksum++)
    {
        if (pinGPIO4.Read() == GpioPinValue.High)
        {
            valoreLettoDalPin = 1;

            valoreChecksum |= (valoreLettoDalPin << idxChecksum);
```

```
            #if PRINT
                Debug.Write("1");
            #endif
        }
        else
        {
            valoreLettoDalPin = 0;

            valoreChecksum |= (valoreLettoDalPin << idxChecksum);

            #if PRINT
               Debug.Write("0");
            #endif
        }

        delay(ritardoBitMicroSecondi);
    }

    return valoreChecksum;
}

int calcoloDelChecksumSulDatagramma(
                                        int bitsDirezioneVento1,
                                        int bitsVelocitaVento1
                                    )
{
    int valoreChecksum = 0;

    valoreChecksum += bitsDirezioneVento1 & 15;
    valoreChecksum += bitsVelocitaVento1 & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 4) & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 8) & 15;
```

```
            return valoreChecksum;
        }

        void delay(int microSecondiDiRitardo)
        {
            int conteggioCicli = 0;

            Stopwatch timerDiPrecisione = Stopwatch.StartNew();

            long ticksIniziale = timerDiPrecisione.ElapsedTicks;

            int ticksNecessari = (int)(
                                        (double)microSecondiDiRitardo *
                                        ((double)Stopwatch.Frequency / 1000000.0)
                                    );

            while (timerDiPrecisione.ElapsedTicks < (ticksIniziale + ticksNecessari))
            {
                conteggioCicli++;
            }

            timerDiPrecisione.Stop();
        }

    }
}
```

4.2.6 Progetto UWP C++

Il capitolo chiude con la realizzazione del medesimo progetto in linguaggio C++. E' interessante notare l'implementazione del metodo "delay()" che richiama le API "QueryPerformanceCounter()" e "QueryPerformanceFrequency()". Vista la vastità di questo linguaggio non gestito si riporta esclusivamente il codice senza entrare nel dettaglio implementativo delle singole istruzione che, comunque, ricalcano la medesima logica di quanto visto in C#.

4.2.6.1 MainPage.xaml.h

```
#pragma once
#include "MainPage.g.h"
#include <string>
#include <vector>
using namespace Windows::Devices::Gpio;
using namespace Windows::UI::Core;
using namespace std;

namespace StazioneMeteo2017_C__
{
    public ref class MainPage sealed
    {
        private:
            GpioPin^ pinGPIO4;
            GpioPin^ pinGPIO17;
            GpioController^ gpio;
            Windows::Foundation::EventRegistrationToken cookie;
             int64 TicksASecond;
             const double ritardoBit = 1.22;
             const double ritardoDatagramma = 1000;
             const int pinRicezioneTX20 = 4;
             const int pinAttivazioneTrasmissioneTX20 = 17;
```

```
        vector<wstring> elencoDirezioni = { L"N", L"NNE", L"NE", L"ENE",
                                            L"E", L"ESE",  L"SE", L"SSE",
                                            L"S", L"SSW", L"SW", L"WSW",
                                            L"W", L"WNW", L"NW", L"NNW" };

    public:
        MainPage();
        void delay(double milliseconds);
        void attivazioneTX20();
        bool inizializzazioneGPIO();
        void letturaDatiDaTX20(GpioPin^ pin, GpioPinValueChangedEventArgs^ Args);
        int startFrame();
        int direzioneVento1();
        int direzioneVento2();
        int checksum();
        int velocitaVento1();
        int velocitaVento2();
        int calcoloDelChecksumSulDatagramma
                                                (
                                                    int bitsDirezioneVento1,
                                                    int bitsVelocitaVento1
                                                );
    };
}
```

4.2.6.2 MainPage.xaml.cpp

```
#include "pch.h"
#include "MainPage.xaml.h"
#include <string>
#include <stdio.h>
#define PRINT 0

using namespace StazioneMeteo2017_C__;
using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;
using namespace Windows::Devices::Gpio;
using namespace Windows::UI::Core;
using namespace Windows::System;

MainPage::MainPage()
{
    InitializeComponent();

    LARGE_INTEGER ticks;
    QueryPerformanceFrequency(&ticks);
    TicksASecond = ticks.QuadPart;

    attivazioneTX20();
}
```

```
void StazioneMeteo2017_C__::MainPage::attivazioneTX20()
{
    if (inizializzazioneGPIO())
    {
      OutputDebugString(L"Errore inizializzazione GPIO!\r\n");

      return;
    }

    this->pinGPIO17 = this->gpio->OpenPin(this->pinAttivazioneTrasmissioneTX20);
    this->pinGPIO17->SetDriveMode(GpioPinDriveMode::Output);
    this->pinGPIO17->Write(GpioPinValue::High);
}

bool StazioneMeteo2017_C__::MainPage::inizializzazioneGPIO()
{
    this->gpio = GpioController::GetDefault();

    if (this->gpio == nullptr)
        return true;
    else
    {
      this->pinGPIO4 = this->gpio->OpenPin(this->pinRicezioneTX20);
      this->pinGPIO4->SetDriveMode(GpioPinDriveMode::Input);
      this->cookie = this->pinGPIO4->ValueChanged += ref new
                Windows::Foundation::TypedEventHandler<GpioPin^,
                GpioPinValueChangedEventArgs^>(this, &MainPage::letturaDatiDaTX20);

      return false;
    }
}
```

```
void StazioneMeteo2017_C__::MainPage::letturaDatiDaTX20
                                        (
                                            GpioPin^ pin,
                                            GpioPinValueChangedEventArgs^ Args
                                        )
{
    this->Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
            ref new DispatchedHandler([=]()
            {
                if(Args->Edge==GpioPinEdge::FallingEdge)
                {
                    OutputDebugString(L"-------------------------------------------------------\n")
                    this->pinGPIO4->ValueChanged -= this->cookie;

                    int bitsStartFrame=startFrame();
                    int bitsDirezioneVento1=direzioneVento1();
                    int bitsVelocitaVento1=velocitaVento1();
                    int bitsChecksum=checksum();
                    int bitsDirezioneVento2=direzioneVento2();
                    int bitsVelocitaVento2=velocitaVento2();
                    int calcoloChecksum = calcoloDelChecksumSulDatagramma
                                            (
                                                bitsDirezioneVento1,
                                                bitsVelocitaVento1
                                            );

                    if ( bitsStartFrame == 4 && bitsDirezioneVento1 == bitsDirezioneVento2
                        && bitsVelocitaVento1 == bitsVelocitaVento2
                        && calcoloChecksum==bitsChecksum
                       )
                    {
                      _RPT1(0, "La direzione del vento è = %s",
                                        this->elencoDirezioni[bitsDirezioneVento1].c_str());
```

```
                    _RPT1(0, " --- La velocità del vento è pari a %d m/s (%1.f km/h - %1.f
                                mph)\n", bitsVelocitaVento1, (float) bitsVelocitaVento1*3.6,
                                (float) bitsVelocitaVento1*2.2);

                    OutputDebugString(L"-------------------------------------------------------------\n");
                    OutputDebugString(L"/////////////////////////////////////////////////////////////\n");

                    txtDirezioneVento->Text = ref new String(
                                            this->elencoDirezioni[bitsDirezioneVento1].c_str());
                    txtVelocitams->Text = bitsVelocitaVento1.ToString();
                    txtVelocitakmh->Text = ((float)bitsVelocitaVento1*3.6).ToString();
                    txtVelocitamph->Text = ((float)bitsVelocitaVento1*2.2).ToString();
                }

                delay(1000);

                this->cookie = this->pinGPIO4->ValueChanged += ref new
                  Windows::Foundation::TypedEventHandler<GpioPin^,
                  GpioPinValueChangedEventArgs^>(this, &MainPage::letturaDatiDaTX20);
            }
        }));
}

int StazioneMeteo2017_C__::MainPage::startFrame()
{
    int bitsDiStartFrame = 0;

    #if PRINT
        OutputDebugString(L"Start frame = ");
    #endif
```

```
    for (int idxStartFrame = 0; idxStartFrame < 5; idxStartFrame++)
   {
      if (this->pinGPIO4->Read() == GpioPinValue::High)
      {
         #if PRINT
             OutputDebugString(L"1");
         #endif

         bitsDiStartFrame = ((bitsDiStartFrame << 1) | 1);
      }
      else
      {
         #if PRINT
             OutputDebugString(L"0");
         #endif

         bitsDiStartFrame = ((bitsDiStartFrame << 1) | 0);
       }

       delay(this->ritardoBit);
   }

   return bitsDiStartFrame;
}

int StazioneMeteo2017_C__::MainPage::direzioneVento1()
{
   int valoreLettoDalPin = 0;
   int valoreDirezioneVento = 0;

    #if PRINT
        OutputDebugString(L"\nDirezione vento n.1 = ");
    #endif
```

```
    for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
    {
        if (this->pinGPIO4->Read() == GpioPinValue::High)
        {
                valoreLettoDalPin = 1;
                valoreDirezioneVento |= (valoreLettoDalPin<<idxDirezioneVento);

                #if PRINT
                    OutputDebugString(L"1");
                #endif
        }
        else
        {
                valoreLettoDalPin = 0;
                valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

                #if PRINT
                    OutputDebugString(L"0");
                #endif
        }

        delay(this->ritardoBit);
    }

   return valoreDirezioneVento;
}

int StazioneMeteo2017_C__::MainPage::direzioneVento2()
{
    int valoreLettoDalPin = 0;
    int valoreDirezioneVento = 0;
```

```
    #if PRINT
        OutputDebugString(L"\nDirezione vento n.2 = ");
    #endif

    for (int idxDirezioneVento = 0; idxDirezioneVento < 4; idxDirezioneVento++)
    {
       if (this->pinGPIO4->Read() == GpioPinValue::High)
       {
           valoreLettoDalPin = 0;
           valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

           #if PRINT
               OutputDebugString(L"0");
           #endif
       }
       else
       {
           valoreLettoDalPin = 1;
           valoreDirezioneVento |= (valoreLettoDalPin << idxDirezioneVento);

           #if PRINT
               OutputDebugString(L"1");
           #endif
       }

       delay(this->ritardoBit);
    }

    return valoreDirezioneVento;
}
```

```
int StazioneMeteo2017_C__::MainPage::velocitaVento1()
{
    int valoreLettoDalPin = 0;
    int valoreVelocitaVento = 0;

    #if PRINT
        OutputDebugString(L"\nVelocità vento n.1 = ");
    #endif

    for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
    {
        if (this->pinGPIO4->Read() == GpioPinValue::High)
        {
            valoreLettoDalPin = 1;
            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                OutputDebugString(L"1");
            #endif
        }
        else
        {
            valoreLettoDalPin = 0;
            valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

            #if PRINT
                OutputDebugString(L"0");
            #endif
        }

        delay(this->ritardoBit);
    }
```

```
        valoreVelocitaVento &= 511;

        return valoreVelocitaVento;
}

int StazioneMeteo2017_C__::MainPage::velocitaVento2()
{
    int valoreLettoDalPin = 0;
    int valoreVelocitaVento = 0;

     #if PRINT
         OutputDebugString(L"\nVelocità vento n.2 = ");
     #endif

     for (int idxVelocitaVento = 0; idxVelocitaVento < 12; idxVelocitaVento++)
     {
         if (this->pinGPIO4->Read() == GpioPinValue::High)
         {
             valoreLettoDalPin = 0;
             valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

             #if PRINT
                 OutputDebugString(L"0");
             #endif
         }
         else
         {
              valoreLettoDalPin = 1;
              valoreVelocitaVento |= (valoreLettoDalPin << idxVelocitaVento);

              #if PRINT
                  OutputDebugString(L"1");
              #endif
```

```
        }

        delay(this->ritardoBit);
    }

    #if PRINT
        OutputDebugString(L"\n");
    #endif

    valoreVelocitaVento &= 511;

    return valoreVelocitaVento;
}

int StazioneMeteo2017_C__::MainPage::checksum()
{
    int valoreLettoDalPin = 0;
    int valoreChecksum = 0;

    #if PRINT
        OutputDebugString(L"\nChecksum = ");
    #endif

    for (int idxChecksum = 0; idxChecksum < 4; idxChecksum++)
    {
        if (this->pinGPIO4->Read() == GpioPinValue::High)
        {
            valoreLettoDalPin = 1;
            valoreChecksum |= (valoreLettoDalPin << idxChecksum);

            #if PRINT
                OutputDebugString(L"1");
            #endif
```

```
        }
        else
        {
            valoreLettoDalPin = 0;
            valoreChecksum |= (valoreLettoDalPin << idxChecksum);

            #if PRINT
                OutputDebugString(L"0");
            #endif
        }

        delay(this->ritardoBit);
      }

      return valoreChecksum;
}


void StazioneMeteo2017_C__::MainPage::delay(double milliseconds)
{
    LARGE_INTEGER currentTicks;
    QueryPerformanceCounter(&currentTicks);

    double targetTicks = currentTicks.QuadPart + milliseconds*TicksASecond / 1000.0;

     while (currentTicks.QuadPart < targetTicks)
     {
         QueryPerformanceCounter(&currentTicks);
      }
}
```

```
int StazioneMeteo2017_C__::MainPage::calcoloDelChecksumSulDatagramma
                                                    (
                                                        int bitsDirezioneVento1,
                                                        int bitsVelocitaVento1
                                                    )
{
    int valoreChecksum = 0;

    valoreChecksum += bitsDirezioneVento1 & 15;
    valoreChecksum += bitsVelocitaVento1 & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 4) & 15;
    valoreChecksum += (bitsVelocitaVento1 >> 8) & 15;

    return valoreChecksum;
}
```

Capitolo 5

Cloud

Questo capitolo descrive le caratteristiche fondamentali di un cloud e quali possibilità può offrire rispetto ad una classica soluzione client/server. Si elencano alcune soluzioni blasonate presenti oggi sul mercato.

5.1 *I sistemi cloud computing*

Per decenni le soluzioni software basate sul paradigma client/server hanno dominato indisturbate, ma con l'avvento di rete di trasporto sempre più veloci e reti di accesso all'ISP basate sulla fibra, hanno ben presto aumentato le richieste di connessione verso i server, creando dei veri e propri picchi impossibili da gestire. Il cloud mira a risolvere questo problema.

5.1.1 La nascita del cloud

Il paradigma di programmazione client/server ha permesso lo sviluppo e la larga diffusione delle applicazioni distribuite, partendo dalle storiche RPC (Remote Procedure Call) fino a giungere ai Web Services REST, tecnologia ampiamente impiegata sui web server. Il limite tecnologico dei sistemi client/server è la gestione dei picchi di richieste, visto che questa tipologia evidenzia, in generale, problemi di scalabilità. Ovviamente esistono varie tipologie di sistemi server, come i cluster e le batterie di server che lavorano

in modalità round-robin. I costi di queste infrastrutture sono ovviamente molto onerose, quindi non alla portata della piccola media impresa. La scalabilità non è l'unico problema a cui bisogna fare attenzione, visto che la replica dei dati ed il bilanciamento del traffico sono punti essenziali per garantire un servizio perfettamente funzionante. La tecnologia cloud nasce come architettura in grado di dare risposta a tutte queste esigenze, rendendo trasparente la locazione fisica dei files all'utente, visto che è compito dei server presenti nei data center gestire il meccanismo di replica. Oggi giorno le architetture cloud computing più famose sono Microsoft Azure, Dropbox, Xively e Amazon AWS, ma sul mercato esistono molte altre realtà altrettanto valide. Ovviamente lo scopo di questa trattazione non è evidenziare tutte le caratteristiche dei vari prodotti, perché la sola analisi dettagliata di Azure richiederebbe centinaia e centinaia di pagine. I prodotti di cloud computing presenti sul mercato si basano su servizi fondamentali per l'utente e vengono identificati dalle seguenti sigle:

- IaaS (Infrastructure as a Service)
- PaaS (Platform as a Service)
- SaaS (Software as a Service)
- DaaS (Desktop as a Service)
- XaaS (Anything as a Service)

Il modello IaaS si basa sul concetto di infrastruttura hardware virtualizzata all'interno del cloud come VM "Virtual Machine", lasciando l'onere al provider la manutenzione e gestione dei device fisici. Questo permette di avere una forte scalabilità dei servizi installati sulla VM ed il cliente stesso può, in qualsiasi momento, richiedere al gestore del cloud upgrade tecnici della VM in loco. L'utente finale non deve pagare nessuna cifra per i device hardware, così come non si deve preoccupare della dislocazione fisica della VM nel cloud, ottenendo quindi una specie di indipendenza geografica. I costi di utilizzo della IaaS sono legati molto spesso al traffico della VM e vengono dettagliati tramite un report finale.
Il modello PaaS si basa sul concetto di offrire una piattaforma per lo sviluppo di applicazioni e servizi web. In questo modo è possibile creare e testare software lato web e condividerlo con altri team di sviluppo, ma al tempo stesso, creare delle applicazioni che

essendo legate esclusivamente alla piattaforma del cloud, sono indipendenti verso tutti gli altri tipi di device che sfruttano il servizio o l'applicazione stessa. Un classico esempio pratico di PaaS, sono le web app, che permettono la scrittura di un'unica applicazione web la quale funzionerà nel medesimo modo su tutte le varie tipologie di smartphone in commercio, indipendentemente dal sistema operativo che questi mobile phone utilizzano. Grazie alla PaaS si ottiene una sorta di indipendenza software dal sistema operativo del device finale che utilizza l'applicazione nel cloud. Esistono servizi PaaS anche per non sviluppatori, basti pensare i servizi di personalizzazione dei web site con wordpress "one click", che permettono a personale non qualificato di creare delle applicazioni ad hoc secondo le proprie esigenze. Tutti i dati memorizzati dalle applicazioni PaaS vengono gestiti e replicati dal provider del cloud in modo trasparente quindi il cliente finale non deve preoccuparsi di implementare e gestire nessun meccanismo di backup dei dati. Il cloud viene visto sia come sorgente, sia come mezzo di backup.

Il modello SaaS si basa sul concetto di affittare dei servizi web già realizzati ai clienti, come ad esempio Facebook, Twitter, Flicker ecc.., indipendentemente dalla tipologia di dispositivo che richiede il servizio, quindi pc desktop, mobile phone o tablet, sempre garantendo il concetto di indipendenza dallo specifico hardware e dallo specifico sistema operativo utilizzato lato client. Molto spesso questo modello viene indicato come "software on demand" e, in certe tipologie di offerte commerciali, viene richiesto l'acquisto di una licenza d'uso oppure canoni mensili o annuali. Tutti questi servizi web vengono gestiti dallo sviluppatore, quindi eventuali update/upgrade non sono mai a carico dell'utente finale. Il servizio è sempre fruibile in modo indipendente dalla posizione geografica dell'utilizzatore.

Il modello DaaS si basa sul concetto di affittare, tramite una sottoscrizione, l'uso di una postazione desktop con i relativi dati all'utente finale, senza che questi debba preoccuparsi dell'intera infrastruttura di virtualizzazione sottostante, delle patch e bug fixes del sistema operativo e software da installare sulla postazione. L'approccio tecnologico con cui il provider del cloud virtualizza i desktop offerti come servizio è, e deve sempre restare, totalmente trasparente agli utilizzatori del servizio stesso. Oggi giorno molte aziende tendono a tenersi all'interno della compagnia dei server di virtualizzazione come VMware, Hyper-V, ma questo approccio oltre ad essere danaroso sia da un punto di vista hardware, a causa dei server dimensionati per gestire molteplici VM, è anche costoso da un punto di vista software, dato che la piattaforma VMware o Hyper-V ha costi assai elevati ed è

vincolante avere personale per la manutenzione dei server e dei sistemi operativi virtualizzati. Il servizio di clouding DaaS abbatte queste spese rendendo tutto più fluida l'infrastruttura informatica, lasciando le imprese al loro reale business senza doversi preoccupare dei dettagli squisitamente tecnologici.
Il modello XaaS è il pilastro dello IoT "Internet of Things" con lo scopo di gestire, memorizzare e monitorare tutta una rete di sensori presenti in ambito aziendale, ma anche in ambito pubblico, come telecamere, trasduttori di pressione per il traffico, stazioni meteo e molto altro. Questo modello di cloud è sicuramente la punta di diamante della nuova era tecnologica, col fine di rendere indipendenti un pool di sensori che di per se sono dispositivi di rilevamento nudi e crudi. Allo XaaS è possibile associare uno strato software che mira a trasformare i device IoT in dispositivi intelligenti, grazie al concetto di "Machine Learning". In questa trattazione non verrà affrontata la tematica del "Machine Learning" che richiederebbe, ovviamente, una discussione a parte.
Una volta discusse le tipologie di modello di cloud computing, resta all'utente capire le sue reali necessità e scegliere quindi un provider ad hoc al quale affidare il proprio business. I servizi di cloud computing più famosi, come accennato in precedenza, sono Azure di Microsoft, Dropbox della ononima Dropbox Inc., Xively della Logme Inc e Amazon AWS, ma esistono anche altre realtà altrettanto valide solo meno blasonate.

5.1.2 Dropbox

Il cloud Dropbox è probabilmente il modello di SaaS più utilizzato al mondo, visto la possibilità di memorizzare file in modo gratuito fino ad uno spazio massimo di 3GB. In tale ottica molto spesso l'acronimo SaaS viene modificato da "Software as a Service" in "Storage as a Service", visto che il cliente finale può memorizzare in modo semplice i propri files sia tramite browser, sfruttando il drag&drop, sia tramite apposita applicazione da installare sul proprio calcolatore, software che viene rilasciato per varie piattaforme. La memorizzazione di files non è però l'unica possibilità offerta da questo modello di cloud, infatti vengono offerte una serie di API tramite le quali lo sviluppatore può scrivere delle applicazioni in locale su device IoT, col fine di inviare i dati nel cloud. Questo approccio di XaaS gratuito rende l'uso di Dropbox consigliato per monitorare i risultati finali prodotti da una rete di sensori casalinga, ma anche aziendale. La Fig.264 mostra l'accesso al sito ufficiale di Dropbox nel quale si può procedere alla registrazione gratuita, oppure optare per un account business con tariffa annuale.

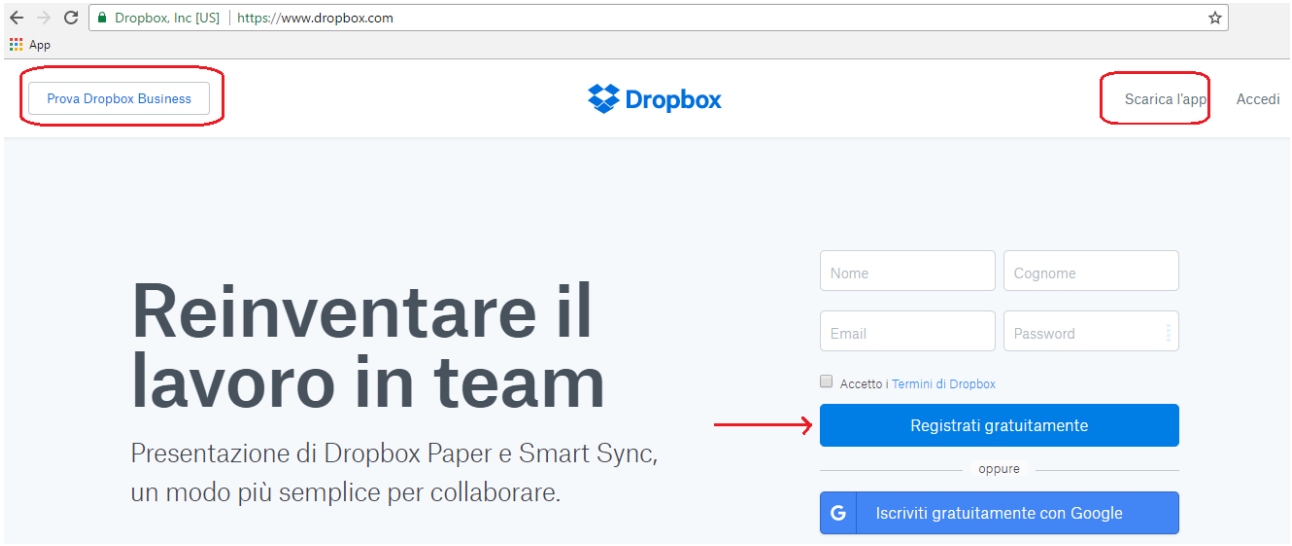


Fig.264

Nella sezione "Scarica l'app" si può scaricare il software che permette il trasferimento dei dati utilizzando il file manager del sistema operativo della postazione client, come si vede dalla Fig.265.

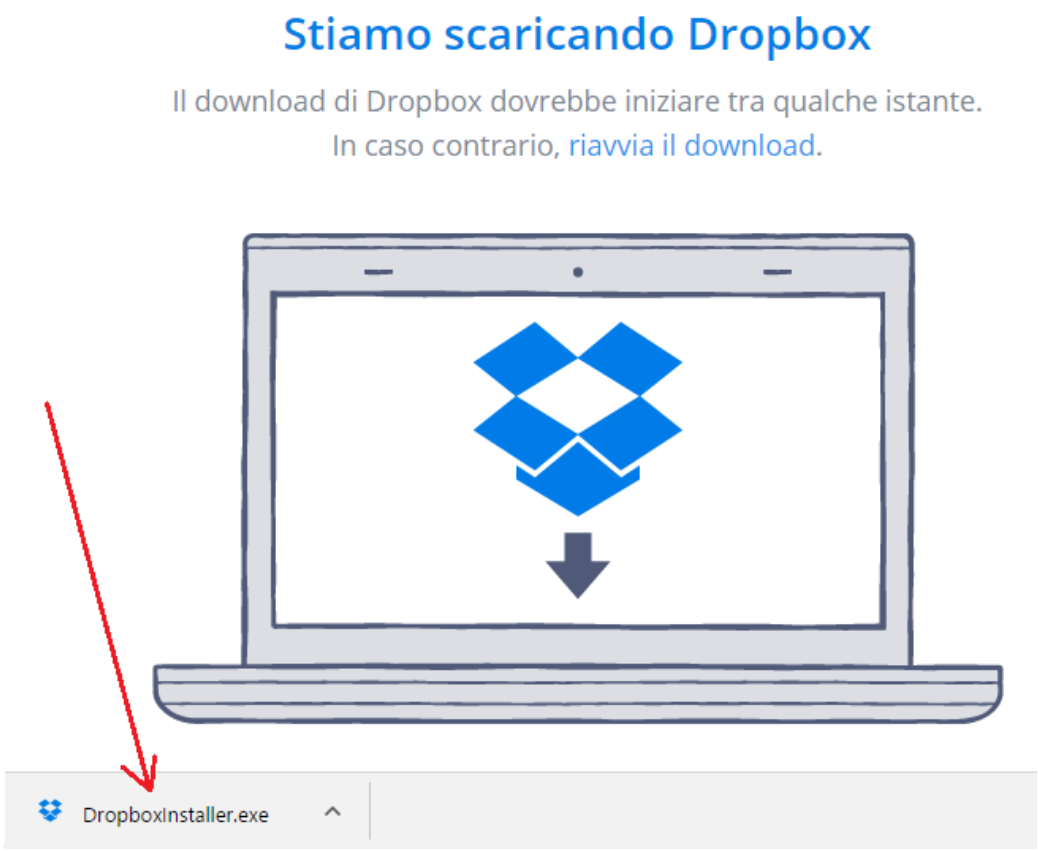


Fig.265

5.1.3 Xively

Il cloud Xively è un esempio di XaaS per la visualizzazione in tempo reale dei dati provenienti da sensori, mostrandoli sotto forma di grafico, quindi un design molto comodo da consultare anche per l'utenza non molto esperta. I dati dei sensori possono anche essere raggruppati e visualizzati assieme, così da rendere più compatto il grafico e più facile da leggere. Un esempio potrebbe essere direzione e velocità del vento, temperatura e pressione atmosferica ecc..Per memorizzare i dati nel cloud Xively, il provider Loginme Inc. offre delle API per una serie di linguaggi di programmazione, permettendo di registrare un canale dedicato al device IoT che invia i dati, con conseguente visualizzazione in modalità grafico lato web. Ovviamente è necessario prima creare un account, che può essere gratuito oppure business. E' lasciato al lettore provare gli step per utilizzare questo tipo di XaaS cloud, così come nel caso di Dropbox. La Fig.266 riporta la home page di questo prodotto.

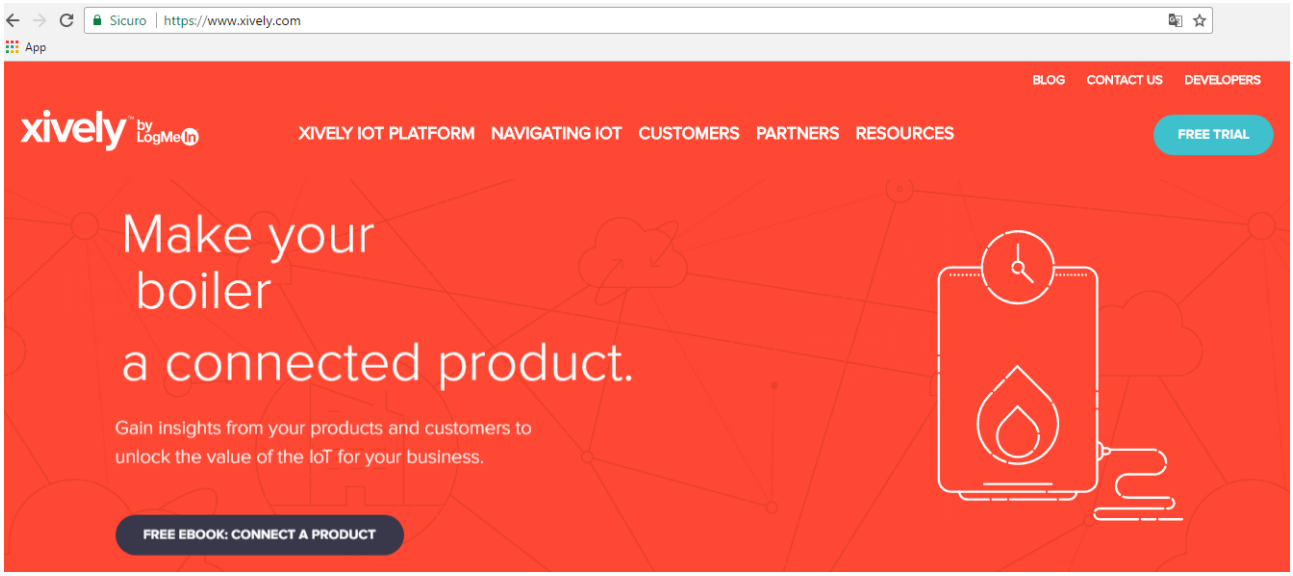


Fig.266

Questi due prodotti sono molto interessanti e facili da utilizzare, anche se il set di API potrebbe essere diverso a seconda dell'impiego di un linguaggio gestito come Java/C# o non gestito come C/C++. Realizzare la stazione meteo con questi prodotti potrebbe comportare l'abbandono di Windows 10 IoT, per il quale al momento non sono state rilasciate ancora delle specifiche API, a favore di una piattaforma Linux, ottima e altrettanto robusta per questi fini. La scelta del cloud si rivela quindi fondamentale.

5.1.4 Amazon AWS

Il cloud Amazon AWS "Amazon Web Services" è un prodotto completo, visto che come Microsoft Azure, offre tutte le funzionalità discusse nel paragrafo 5.1.1, quindi è una valida scelta a cui ogni utente finale dovrebbe pensare. Un scelta che senza ombra di dubbio premia Azure nei confronti di AWS, è l'impiego della versione gratuita di Visual Studio 2015/2017 Community Edition, che permette allo sviluppatore di interagire direttamente con il cloud Azure, anche se la controparte di sviluppatori Java molto probabilmente trova più alettante appoggiarsi ad AWS che Azure. Come sempre le API a disposizione del linguaggio scelto per le proprie applicazioni sono un vincolo nella scelta del cloud. Vista la vastità di AWS è impensabile iniziare una qualsiasi trattazione di questo ottimo prodotto, anche perché la scelta dello scrivente nell'impiego del cloud ricade su Microsoft Azure. Nulla vieta al lettore di cimentarsi nell'uso di AWS per inviare i dati dal Pi 3 nello storage di Amazon, visto la presenza del supporto ai device IoT. In Fig.267 Amazon AWS.

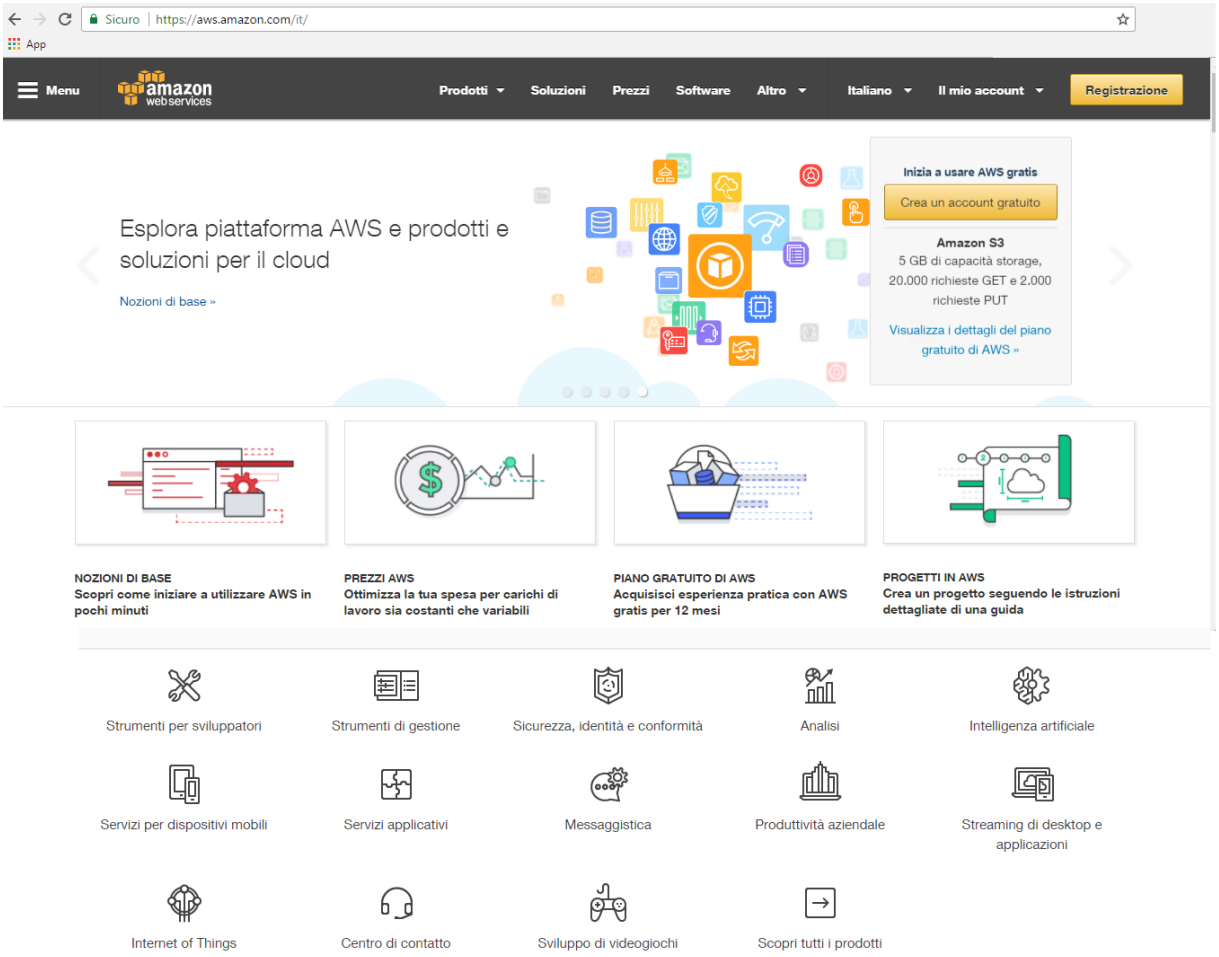


Fig.267

5.2 *Microsoft Azure*

Prodotto di punta nel mercato delle piattaforme cloud computing, sistema che offre tutte le funzionalità che un'azienda necessità, dalla virtualizzazione dei device, allo storage, dalla gestione dello streaming IoT, all'analisi del traffico e molte altre cose. Il tutto viene accompagnato da una serie di API per i linguaggi integrati nella suite Visual Studio 2015/2017 come C++, C#, Javascript e Python, anche se per quest'ultimo non è ancora possibile creare interfacce grafiche con Visual Studio. Il programmatore può scaricare i pacchetti che necessità dal Visual Studio stesso, sfruttando il packet manager "NuGet". Gli aspetti relativi allo sviluppo software verranno tratti nelle sezioni successive, in questo paragrafo viene presentato Azure e descritti i servizi, in sintesi, che verranno utilizzati per permettere il salvataggio dei dati nello storage del cloud e la successiva visualizzazione. Inutile dire che una trattazione dettagliata di Azure richiederebbe più di un libro ad hoc.

5.2.1 Sottoscrizione gratuita 30 giorni

La sottoscrizione ad Azure è un passo iniziale fondamentale, senza la quale non è possibile collegarsi al portale del cloud ed iniziare ad utilizzare i servizi desiderati. La politica di Microsoft è legata al traffico, quindi mensilmente viene prodotta una bolletta di spesa legata si al traffico, ma anche alle configurazioni dei servizi in termini di replica dei dati, memoria, bilanciamento e molti altri parametri. Una corretta configurazione del tipo di servizio voluto è fondamentale per non arrivare a spese elevate senza giusta motivazione. Fortunatamente esiste una sottoscrizione completa valida 30 giorni, con la quale l'utente può provare tutti i servizi del cloud senza pagare nulla. Terminati i 30 giorni la sottoscrizione viene disabilitata e l'utente può eventuale passare ad una sottoscrizioni a pagamento senza perdere i servizi che ha precedentemente configurato. Esiste anche una sottoscrizione per gli studenti della scuole superiori e delle università legata al programma Microsoft Imagine, solo che non tutti i servizi offerti da Azure possono venire utilizzati gratuitamente, infatti il servizio "Azure IoT Hub" non è compreso in questa tipologia di sottoscrizione, cardine fondamentale per il progetto in questione. In definitiva si utilizza una sottoscrizione gratuita, convertita poi a pagamento così da mostrare al lettore che tutto resta invariato. Esistono anche altre sottoscrizioni oltre a quella free, a pagamento e legata al programma Microsoft Imagine. Una di queste dipende dal MSDN per sviluppatori, quindi aziende che sottoscrivono questo pacchetto potranno sviluppare e testare le

proprie applicazioni senza pagare costi relativi al traffico. Il prodotto commerciale venduto sarà però soggetto al pagamento del traffico. Il collegamento al sito ufficiale di Azure avviene tramite la URL https://azure.microsoft.com come riportato in Fig.268.

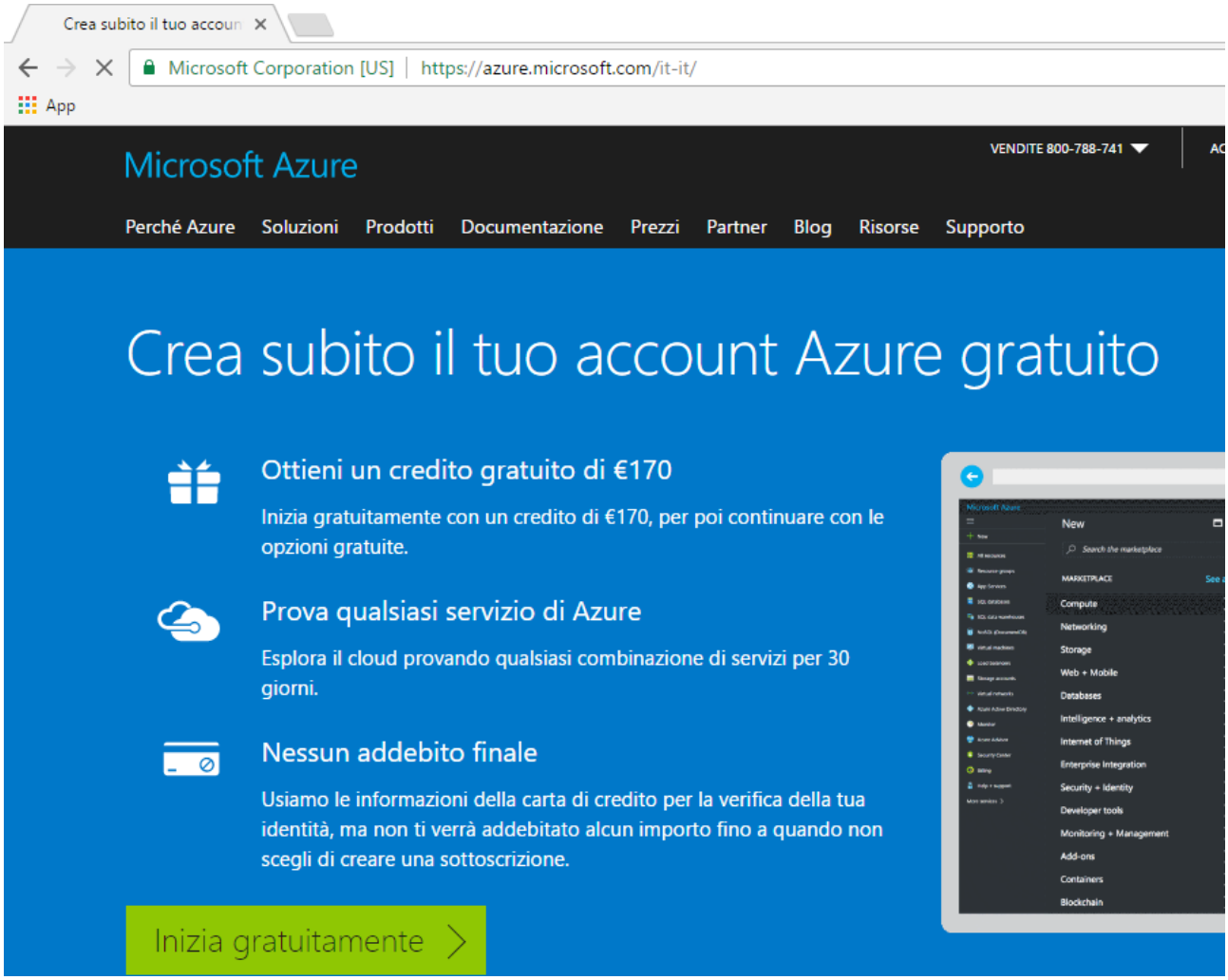


Fig.268

Procedere alla fase di registrazione premendo il pulsante "Inizia gratuitamente" tramite il quale viene richiesto di inserire le credenziali di un account Microsoft precedentemente creato. Nel caso in cui l'utente non disponga di un account Microsoft, può procedere alla creazione cliccando sul link "Crea un nuovo account Microsoft" e può optare per creare un nuovo account di posta legato ai domini Microsoft o utilizzare un account di posta personale. La scelta è ovviamente dell'utente. In Fig.269 è riportata la schermata di autenticazione.

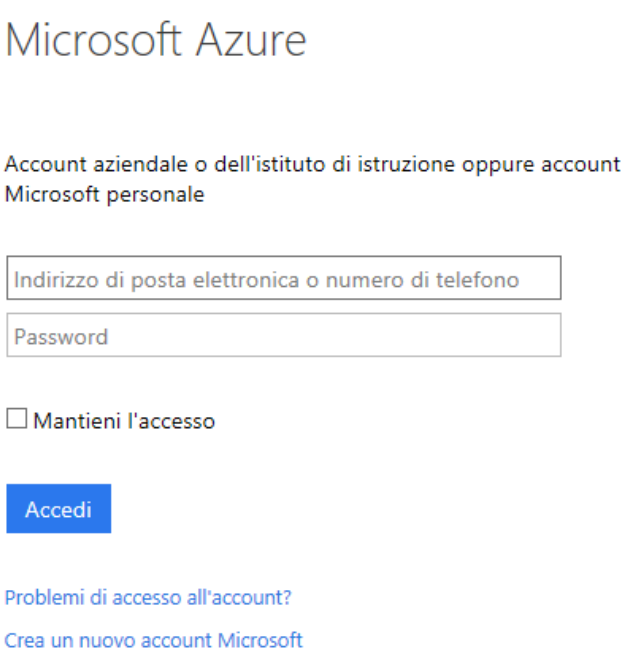


Fig.269

L'account di posta è obbligatorio per continuare la fase di registrazione al portale Azure, come si vede chiaramente nella Fig.270.

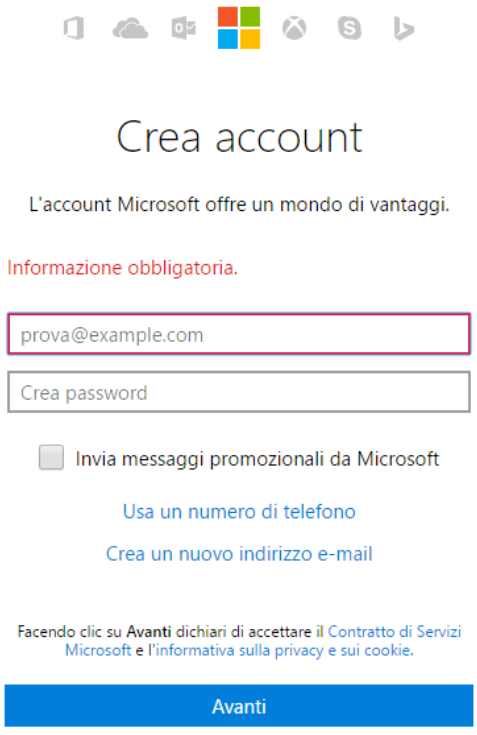


Fig.270

Ogni sottoscrizione di 30 giorni dispone di un credito Azure di 170€, cifra che se non viene spesa per intero non può in nessun modo venire stornata su altre tipologie di sottoscrizioni. Questa scelta dovrebbe spingere l'utente a provare tutti i servizi offerti. Il primo step è la registrazione delle informazioni personali, come si evince dalla Fig.271.

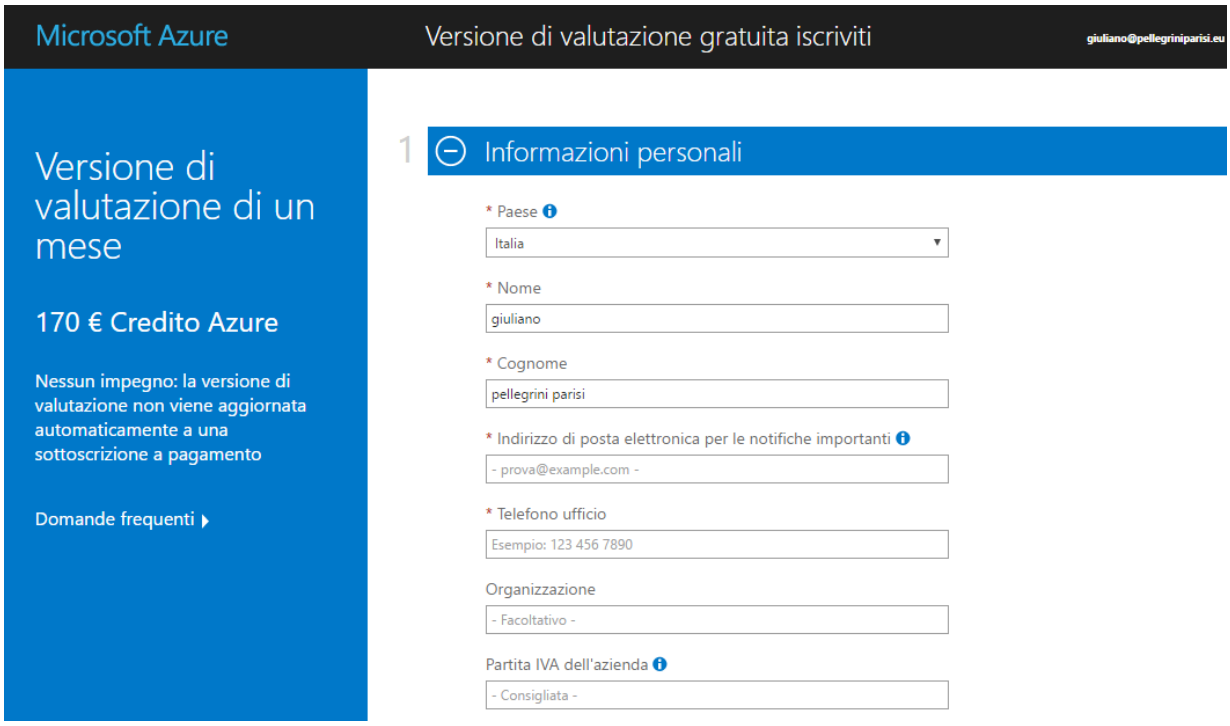


Fig.271

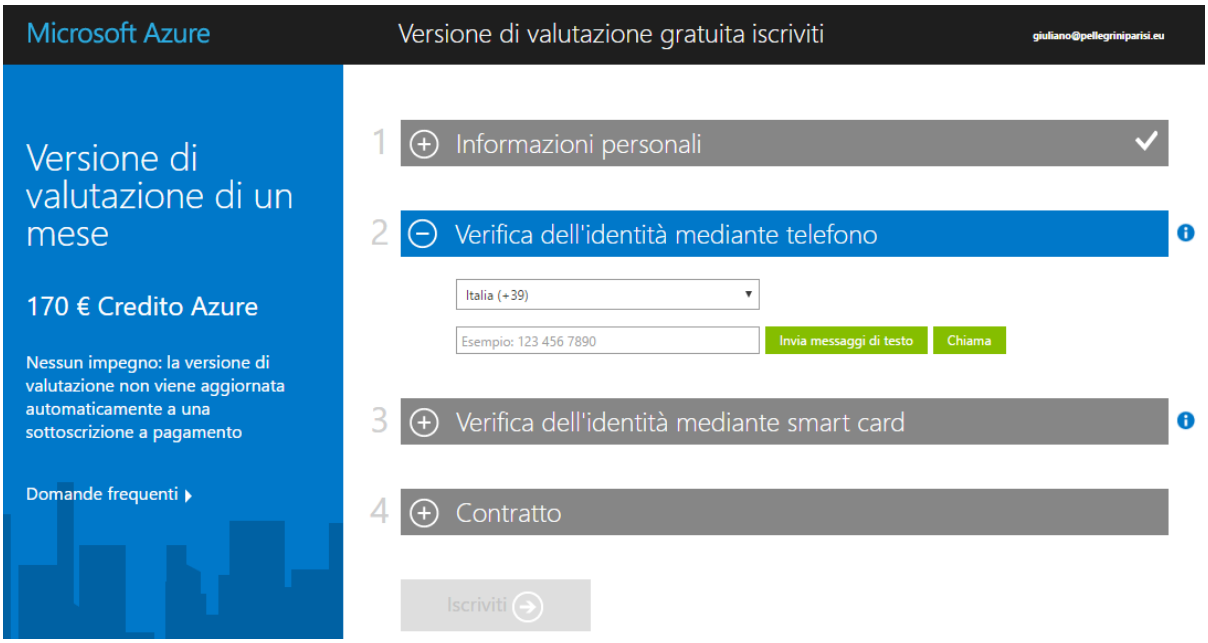


Fig.272

In Fig.272 l'utente dovrà inserire le credenziali telefoniche e optare per esempio nell'invio di un codice di conferma del numero cliccando sul pulsante "Invia messaggio di testo". La Fig.273 riassume questo operazione.

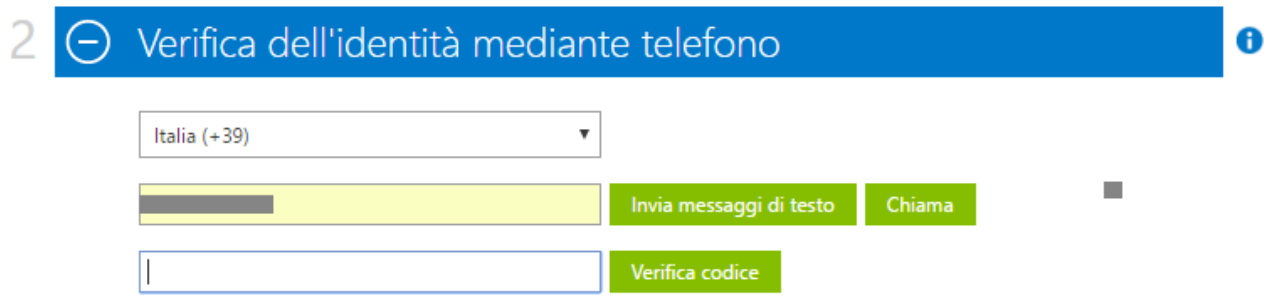


Fig.273

Una volta ricevuto il codice, va inserito nell'apposita sezione e confermato tramite il pulsante "Verifica codice", in modo da passare alla sezione successiva di Fig.274.

Fig.274

La sottoscrizione è gratuita, ma la Microsoft richiede ugualmente che l'utente inserisca una carta di credito valida. Al momento della sottoscrizione lo scrivente ha verificato che le carte di credito ricaricabili non sono accettate, politica che però potrebbe in futuro venire modificata dalla Microsoft. Terminato l'inserimento della carta, non resta che confermare il contratto della sottoscrizione premendo il pulsante "Iscriviti" di Fig.275.

Fig.275

La sottoscrizione è terminata e non resta che accedere al portale Azure come da Fig.276.

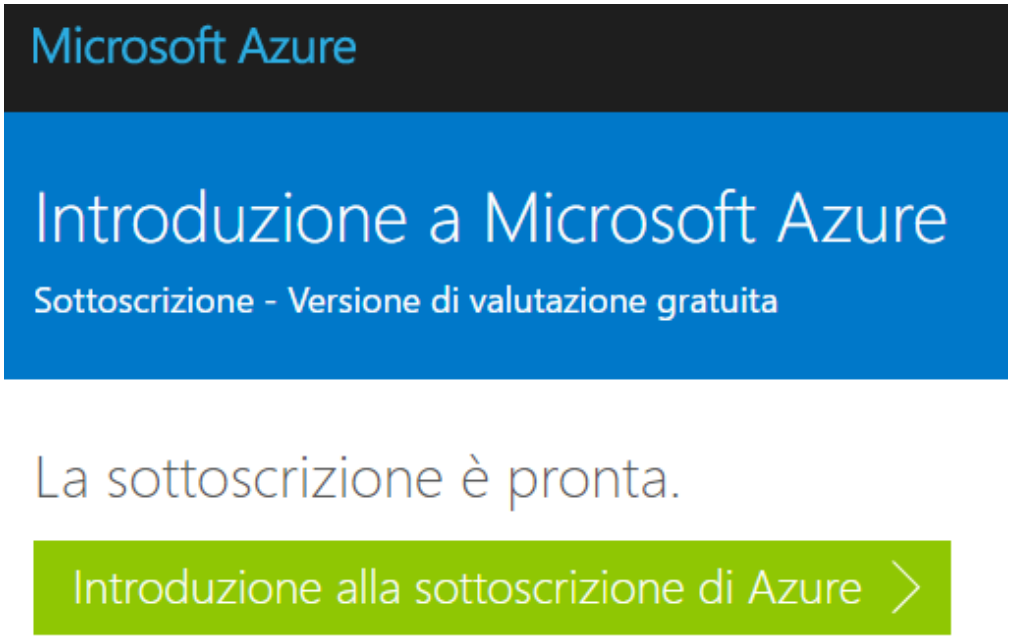


Fig.276

5.2.2 Creazione dello storage

Terminata la sottoscrizione, non resta che entrare, previa autenticazione, nel portale azure https://portal.azure.com, selezionando l'account della sottoscrizione creata in precedenza. Nulla vieta di avere più account azure, quindi sarà necessario selezionare quello legato alla sottoscrizione che si desidera utilizzare senza immettere la password. La Fig.277 riporta questo step.

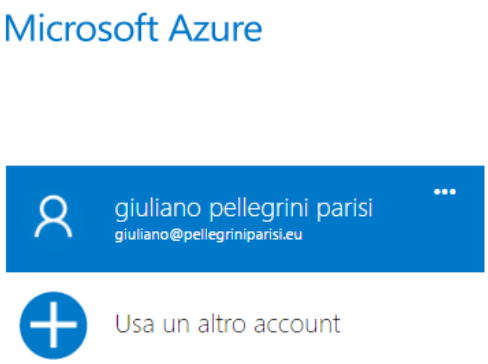


Fig.277

Successivamente il meccanismo automatico di ridirezione cambia la pagina con successiva richiesta di inserimento password come si evince dalla Fig.278.

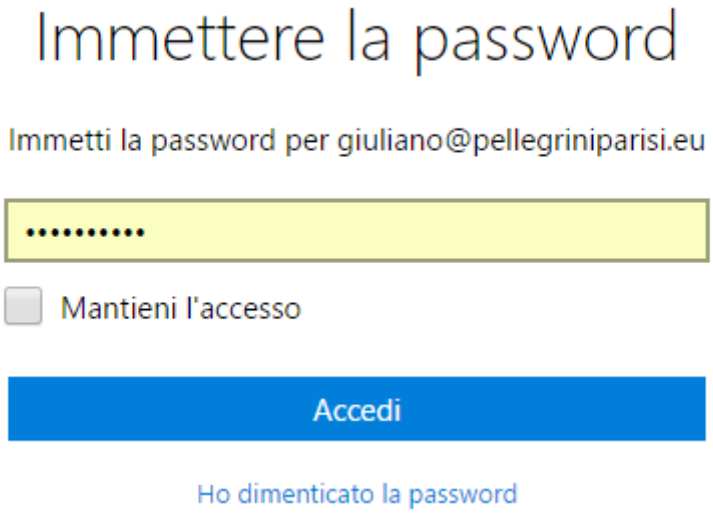


Fig.278

Una volta effettuato l'accesso si presenta la dashboard del cloud Azure, mentre sulla colonna di sinistra è possibile selezionare i servizi che si vogliono utilizzare. La Fig.279 mostra il primo accesso al portale.

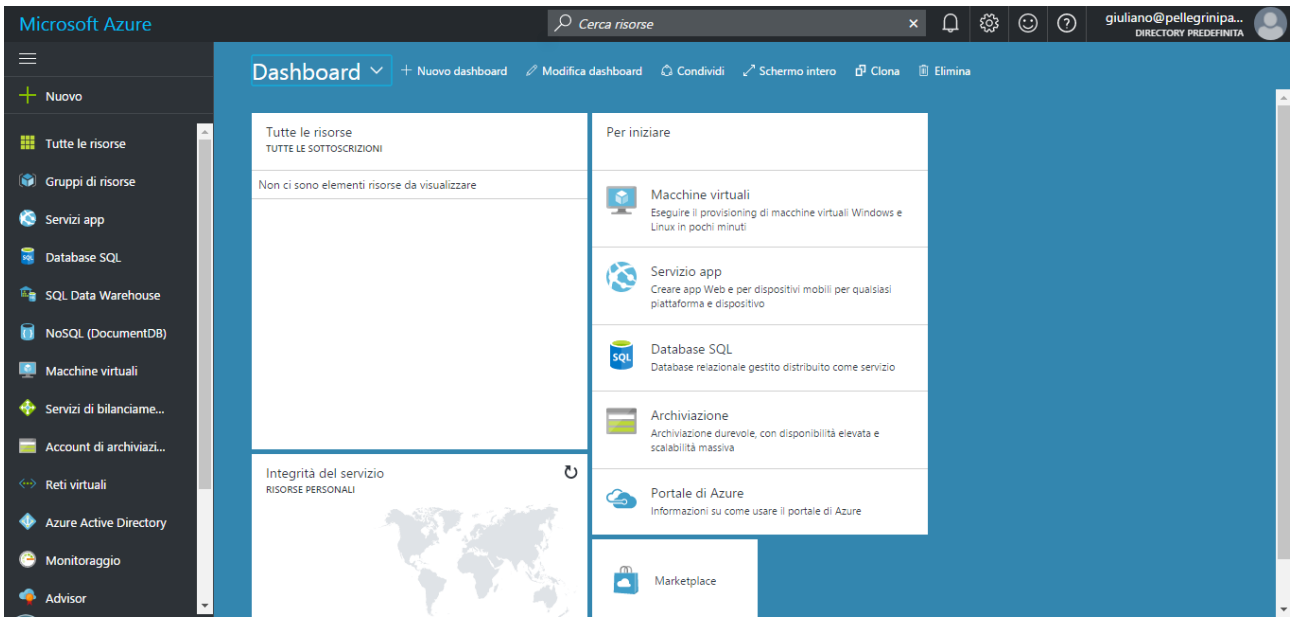


Fig.279

Il primo servizio che è necessario implementare è il contenitore che servirà per memorizzare i dati che arriveranno dall'applicazione UWP C# presente sul Raspberry Pi 3. Premere sulla voce "+ Menu" in modo da procedere all'aggiunta del servizio, selezionando poi "Storage" e poi "Storage account - blob file, table, queue" come da Fig.280.

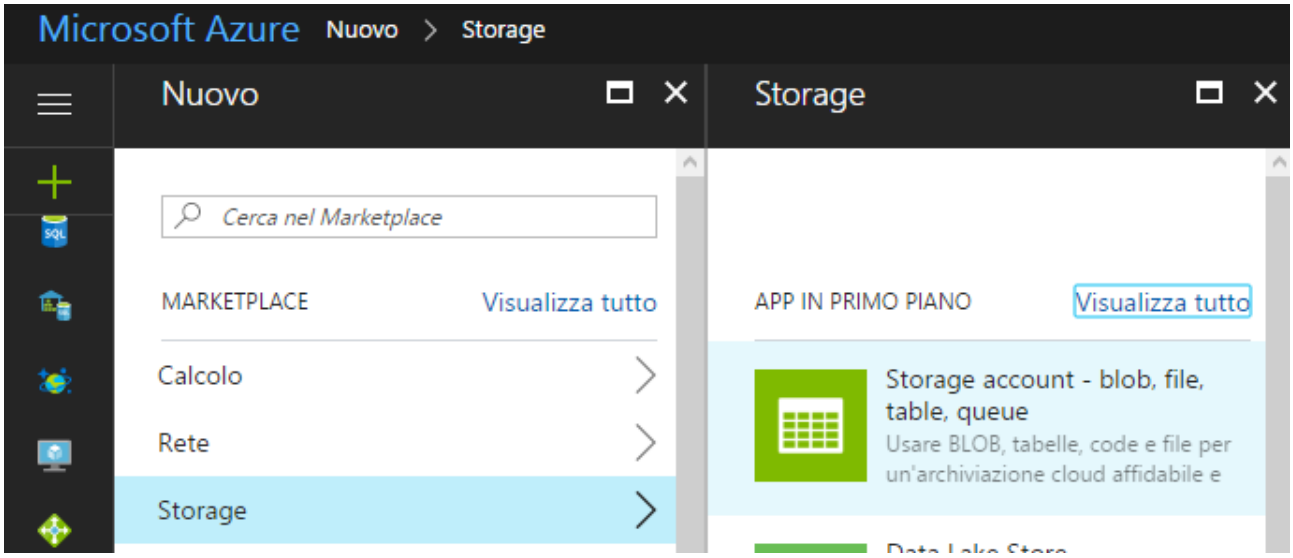


Fig.280

Indicare come nome di archiviazione "tx20", valore che sarà poi utilizzato lato software, visto che nulla vieta di avere diverse tipologie di storage. In tale contesto si opta per un'archiviazione di tipo BLOB, visto che le informazioni relative quali velocità e direzione del vento, verranno inviate allo storage dalla UWP C# dopo un procedimento di serializzazione JSON. Lasciare i valori di default come da Fig.281.

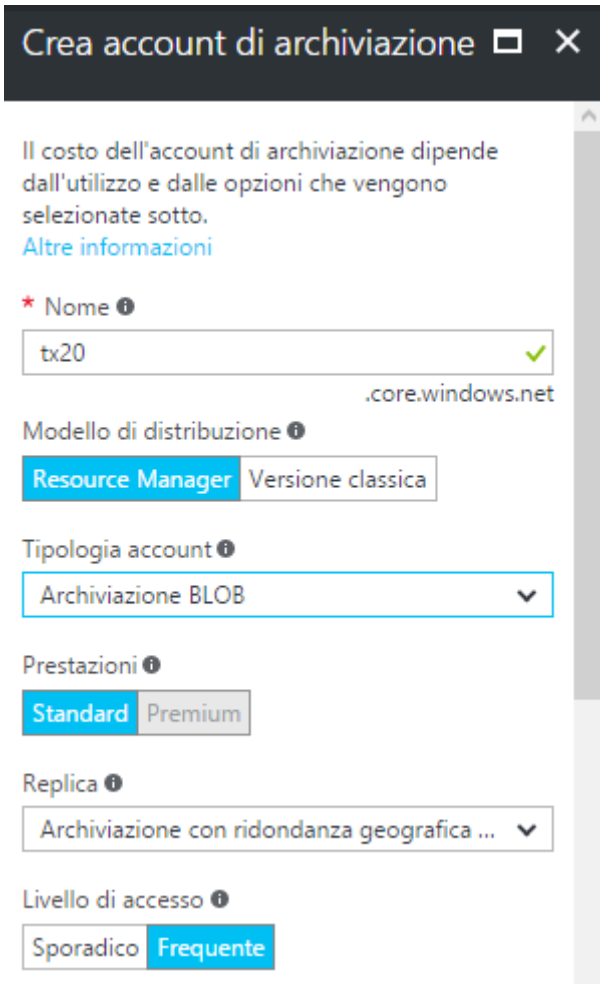


Fig.281

E' interessante notare il meccanismo di replica dei dati, che avviene con un'archiviazione con ridondanza geografica, ossia i dati vengono copiati sui server presenti nei data center dislocati in altre parti del mondo. L'utente finale non dovrà pensare a nessun tipo di backup aziendale di questi dati. E' possibile incrementare le prestazioni del servizio di storage selezionando "Premium", ovviamente questa modifica comporta un costo mensile maggiore. La Fig.282 completa la configurazione indicando la sottoscrizione gratuita ed il nome scelto del gruppo di risorse "Raspberry". Confermare tramite "Crea".

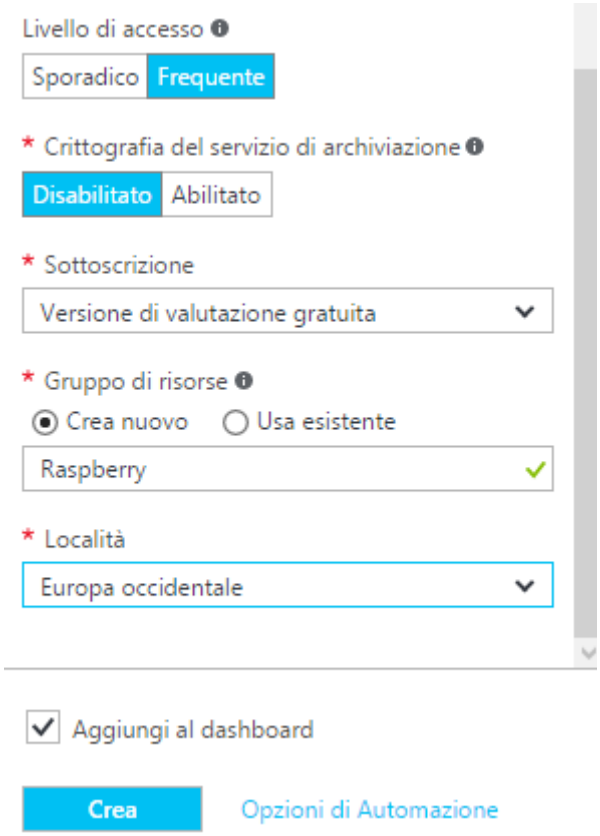


Fig.282

Avendo lasciato spuntata la voce "Aggiungi al dashboard", il servizio viene linkato alla dashboard così che sia più facilmente selezionabile. Si veda la Fig.283.

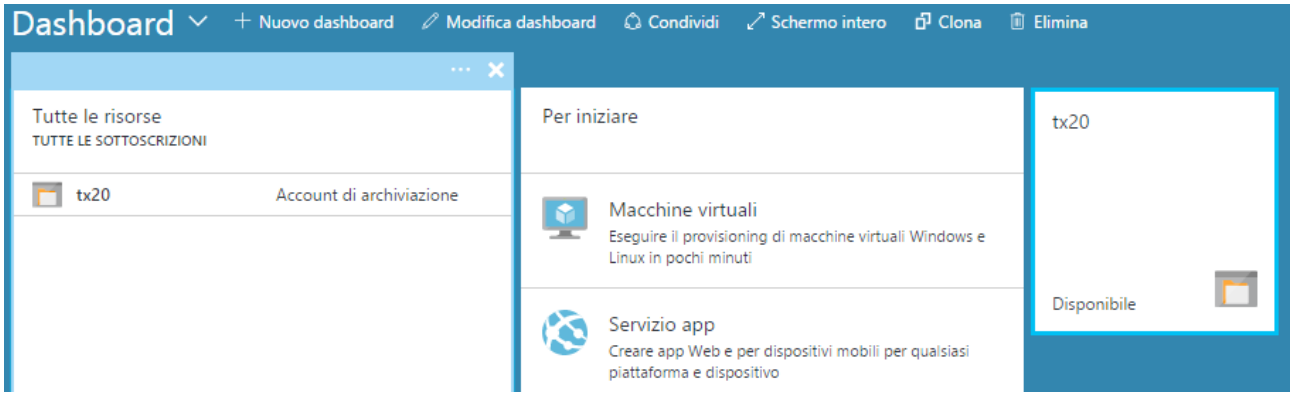


Fig.283

Fino a questo momento però si è implementato un servizio di storage BLOB chiamato "tx20", dentro il quale è necessario inserire uno o più contenitori, che saranno automaticamente di tipo BLOB. In altri termini è possibile all'interno di un unico account di

archiviazione creare più contenitori, con nomi diversi, che servono per memorizzare i dati. Lo sviluppatore lato software deciderà in seguito quali tra questi impiegare. Dalla dashboard selezionare l'account di archiviazione "tx20", come da Fig.284, aggiungendo un contenitore tramite il pulsante "+ Contenitore".

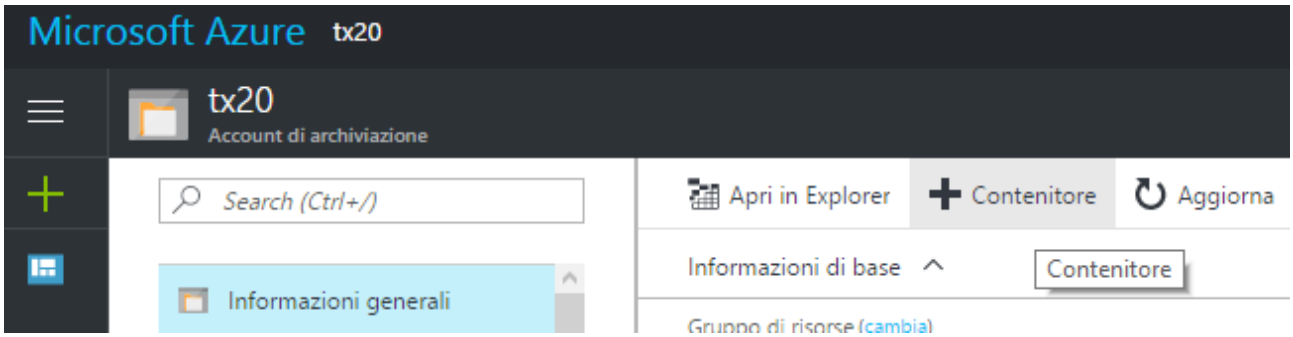


Fig.284

Il nome scelto per l'unico contenitore impiegato in questo progetto è "dati" con un tipo di accesso "Contenitore" come si evince dalla Fig.285.

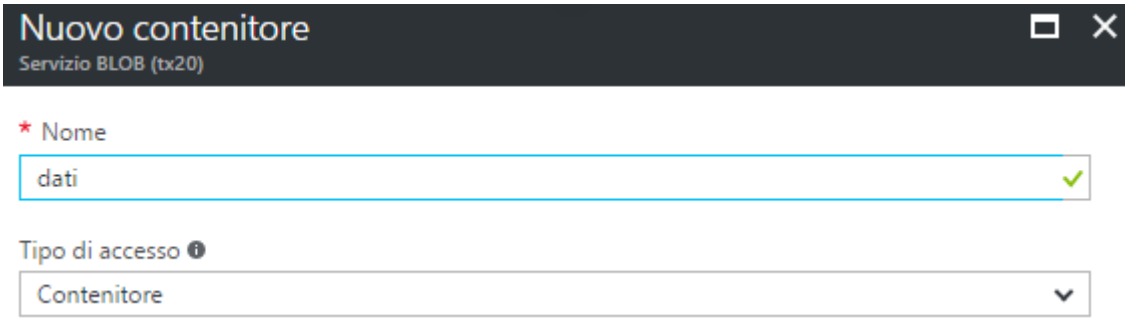


Fig.285

Una volta creato il contenitore viene aggiunto all'interno dell'account di archiviazione, come da Fig.286. Eventuali altri contenitori vengono elencati in questa sezione.

Fig.286

L'utilizzo dell'account di archiviazione "tx20", e quindi la successiva scelta del contenitore, comporta a livello di software l'uso di una connessione di stringa che garantisca l'uso esclusivo del servizio. Lo sviluppatore può copiare direttamente tale stringa accedendo alla sezione "Chiavi di accesso" dell'account "tx20" come da Fig.287.

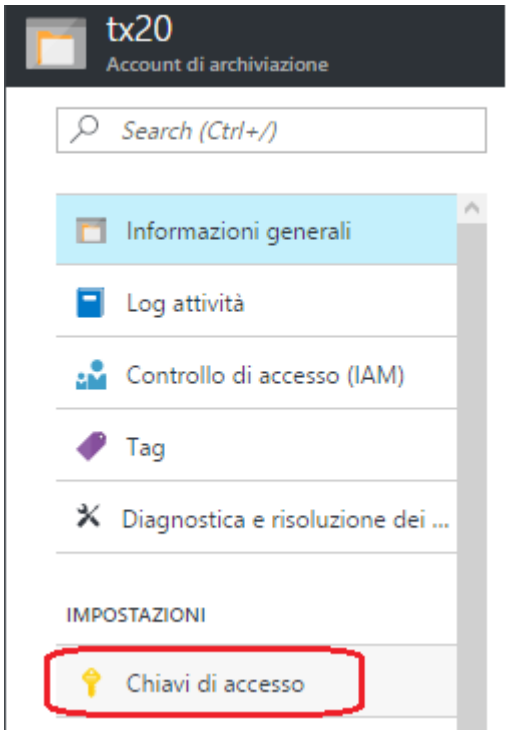


Fig.287

Selezionando l'icona di Fig.288, la stringa di connessione viene copiata nella clipboard di Windows e pronta ad essere incollata all'interno del software.

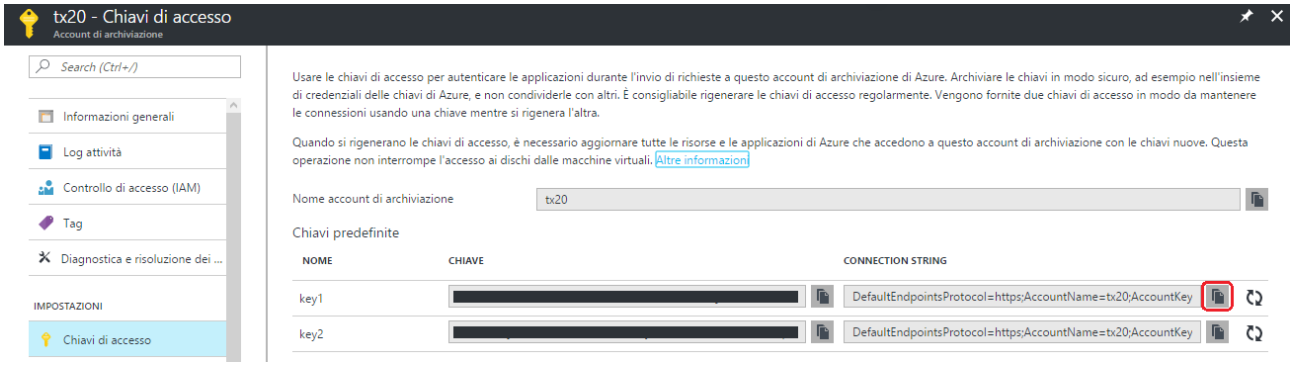


Fig.288

5.2.3 Schema dei servizi

L'account di archiviazione delle informazioni è il primo passo per qualsiasi tipo di applicazione che voglia scrivere/leggere informazioni verso/da Azure. La Fig.289 schematizza il legame necessario tra i vari servizi per l'uso dei device IoT. Una volta creato l'account "tx20", è necessario creare un contenitore BLOB, passo già implementato nella sezione precedente nel quale il contenitore è stato chiamato "dati".

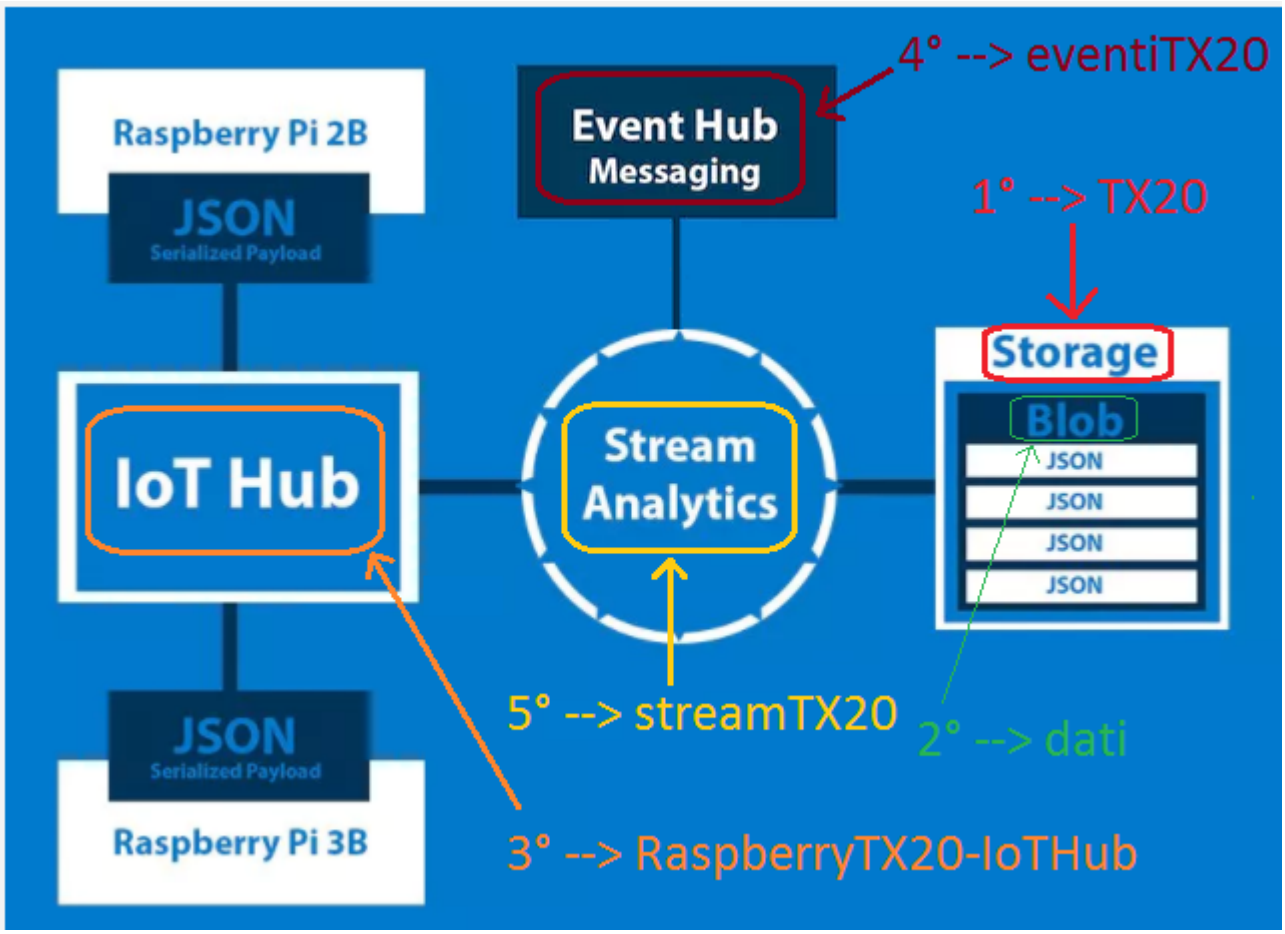


Fig.289

Tutte le informazioni verranno veicolate dal device IoT verso Azure in formato JSON, quindi a livello di applicazione UWP C#, che gira sul Raspberry, conviene utilizzare un meccanismo che serializzi la classe con i dati nel formato JSON. Nel capitolo dedicato all'invio dei dati verso il cloud verrà esposto in dettaglio questo meccanismo. Il passo successivo è creare un Hub per la ricezione dei dati da più possibili device IoT, questo servizio prende il nome di "IoT Hub" e funge, come dice il nome stesso, da tramite tra i device IoT e lo storage dei dati.

Questo servizio di Hub verrà chiamato "RaspberryTX20-IoTHub". Un altro modulo necessario al progetto è il servizio di messaggistica, chiamato "Event Hub Messaging", tramite il quale si annotano sotto forma di evento tutte le informazioni che arrivano dai device IoT, ma si potrebbe anche pensare all'invio di messaggi da Azure ai device IoT, come conferma della ricezione delle informazioni. Questa opzione non verrà implementata nel progetto in questione. Il servizio "Event Hub Messaging" verrà chiamato "eventiTX20". L'ultimo servizio che resta da implementare è l'analizzatore dei dati chiamato "Stream Analytics" con il quale è possibile implementare dei filtri sui dati, secondo specifiche esigenze, prima che questi vengano depositati nel contenitore BLOB. In questo progetto il filtro utilizzato lascerà passare tutte le informazioni verso lo storage. Il servizio "Stream Analytics" verrà chiamato "streamTX20". E' importante ricordare che con la sottoscrizione gratuita 30 giorni è possibile utilizzare tutti questi servizi, mentre con la sottoscrizione Microsoft Imagine per scuole ed università, il servizio "IoT Hub" non è disponibile. Qualsiasi sottoscrizione a pagamento permette l'uso dei vari servizi secondo le specifiche necessità.

5.2.4 IoT Hub

Questo servizio è fondamentale per connettere una moltitudine di dispositivi IoT al cloud Azure. Come indicato in precedenza, verrà chiamato "RaspberryTX20-IoTHub", ma nulla vieta di scegliere un nome più consono ai propri scopi. Per creare il servizio selezionare il pulsante "+ Nuovo" sulla dashboard, accedendo poi alla sezione "Internet delle cose" ed "IoT Hub" come si evince dalla Fig.290.

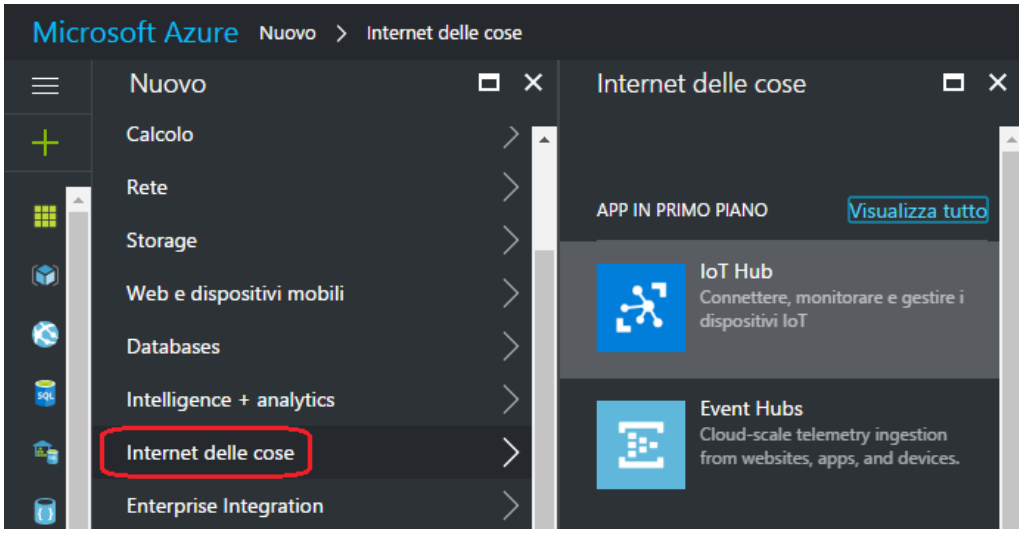


Fig.290

Impostare il nome scelto per il servizio, lasciando il piano tariffario a quello minimo, ossia S1. Il numero di unità nel hub sono "1", mentre il numero delle partizioni resta "4", ossia il default. Utilizzare la sottoscrizione gratuita, il gruppo di risorse "Raspberry" e spuntare l'aggiunta alla dashboard come da Fig.291. Confermare tramite "Crea". In Fig.292 il servizio viene collegato alla dashboard.

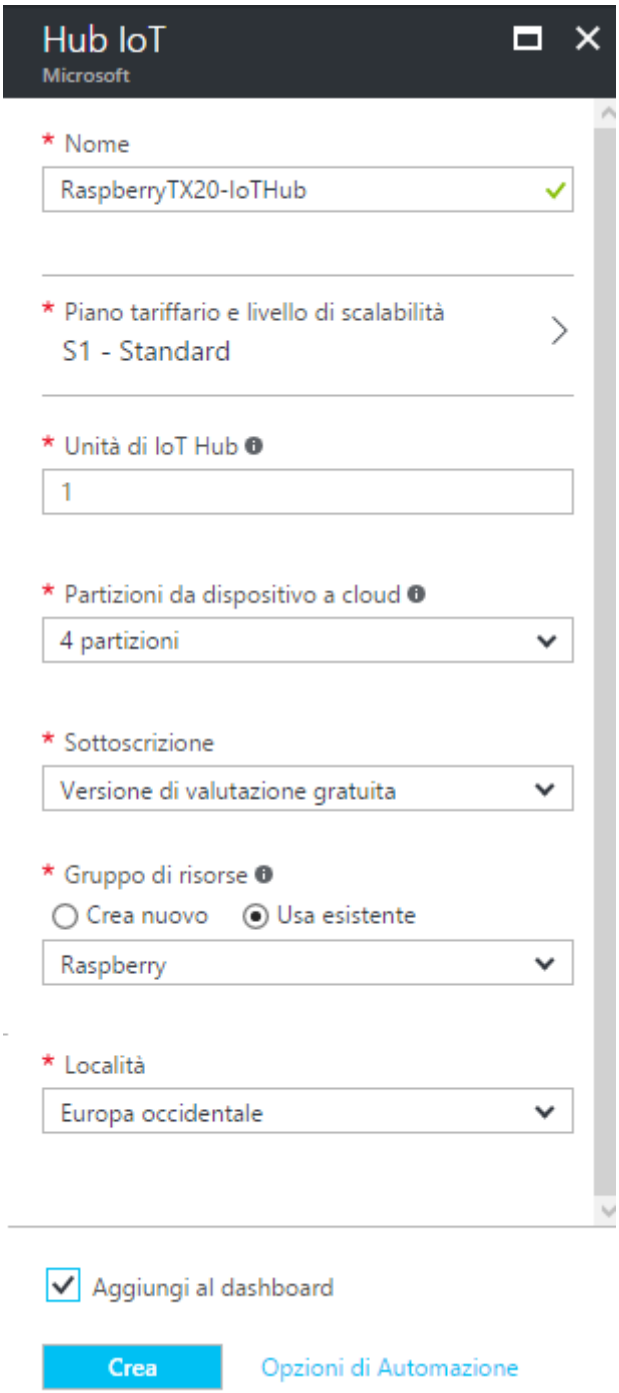


Fig.291

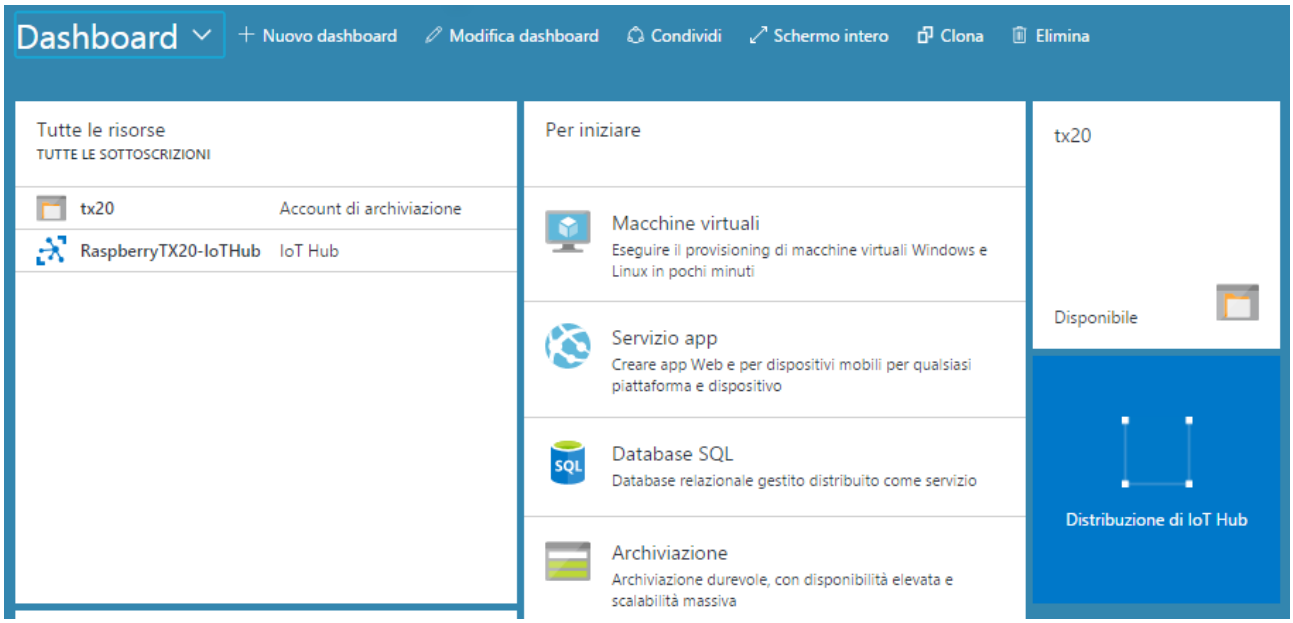


Fig.292

Avendo configurato il servizio "IoT Hub" per gestire un device IoT, è necessaria la creazione del relativo device all'interno del servizio stesso, quindi è necessario selezionare "RaspberryTX20-IoTHub" sulla dashboard e procedere all'inserimento di un device come da Fig.293. L'identificatore del device "Device ID" è "Raspberry1_TX20" con una chiave simmetrica auto generata. Verificare che la voce "Connetti dispositivo all'hub IoT" sia abilitato. Procedere all'inserimento del device tramite il pulsante "+ Aggiungi" come da Fig.294.

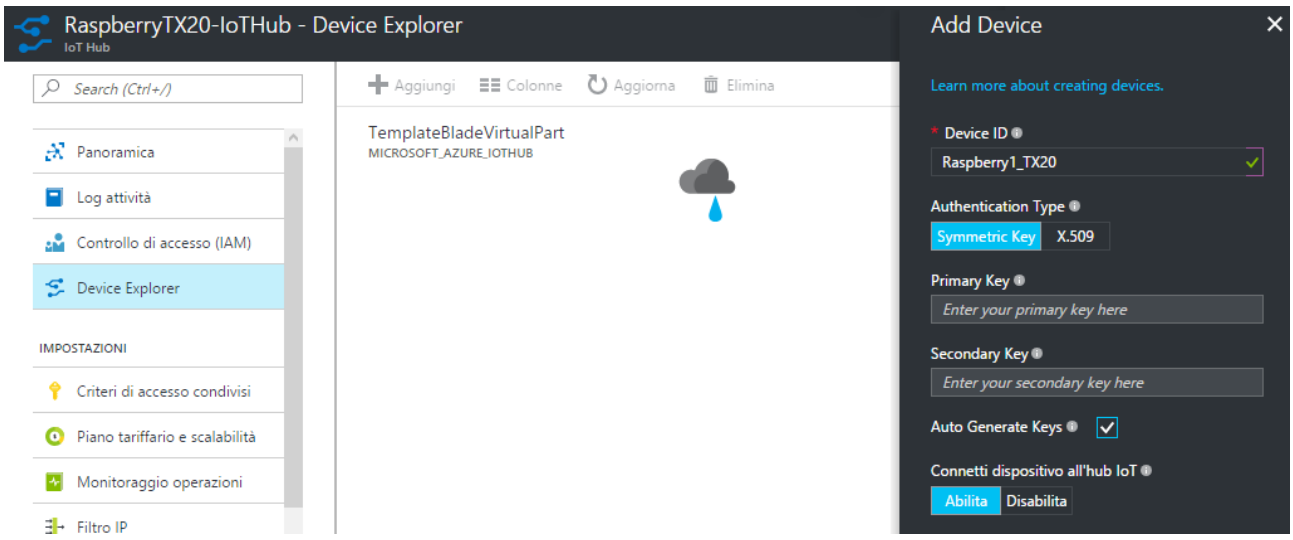


Fig.293

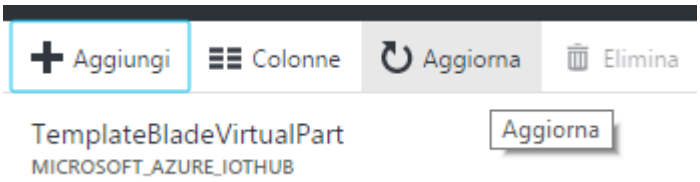


Fig.294

L'aggiunta del device al servizio "IoT Hub" è visibile in Fig.295.

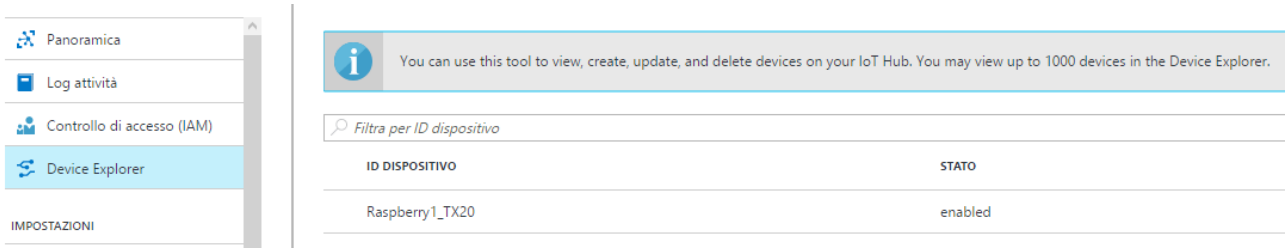


Fig.295

E' fondamentale capire che lato UWP C#, l'accesso a questo servizio passa per il device appena inserito, quindi si dovrà impiegare il nome "Raspberry1_TX20" e come stringa di connessione quella indicata in Fig.296. Per questioni di sicurezza la chiave simmetrica prima e secondaria è stata oscurata in figura, ma è parte integrante della rispettiva stringa di connessione che potrà comodamente venire copiata nella clipboard tramite l'icona in grigetto.

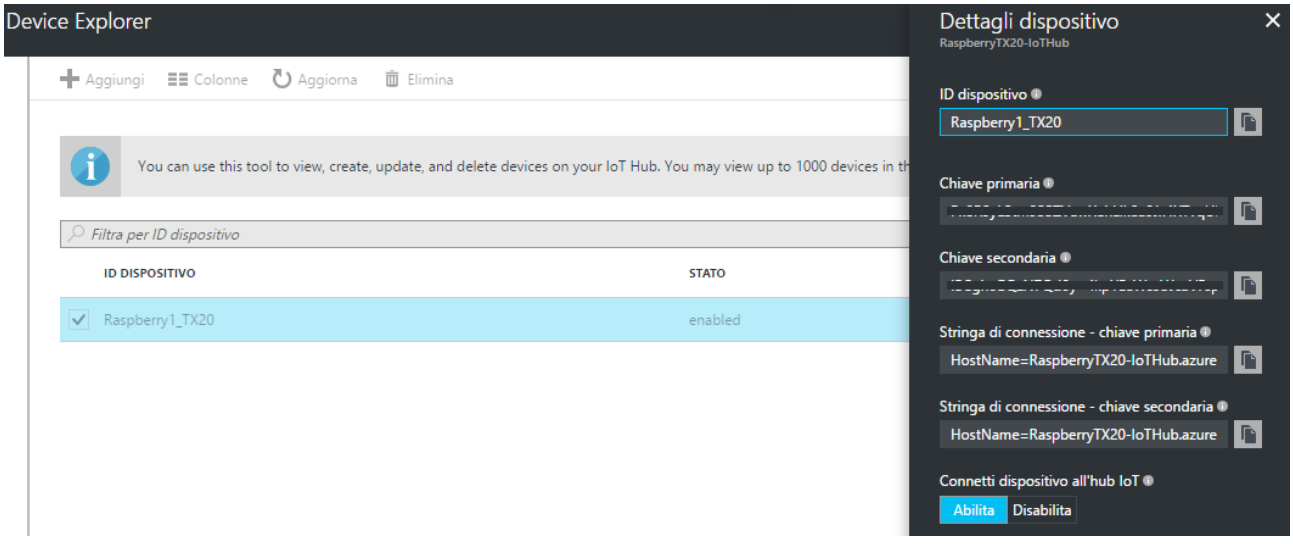


Fig.296

5.2.5 Event Hubs

Questo servizio è fondamentale per catalogare il numero di messaggi in ingresso al cloud e in uscita verso il device IoT. In questo progetto non verrà implementata nessuna forma di conferma di ricezione dati da parte del cloud verso il Pi 3, ma con un minimo di sforzo è facilmente realizzabile. Per aggiungere il servizio utilizzare l'ormai noto pulsante "+ Nuovo" e dal menù "Internet delle cose" selezionare "Event Hubs" come da Fig.297.

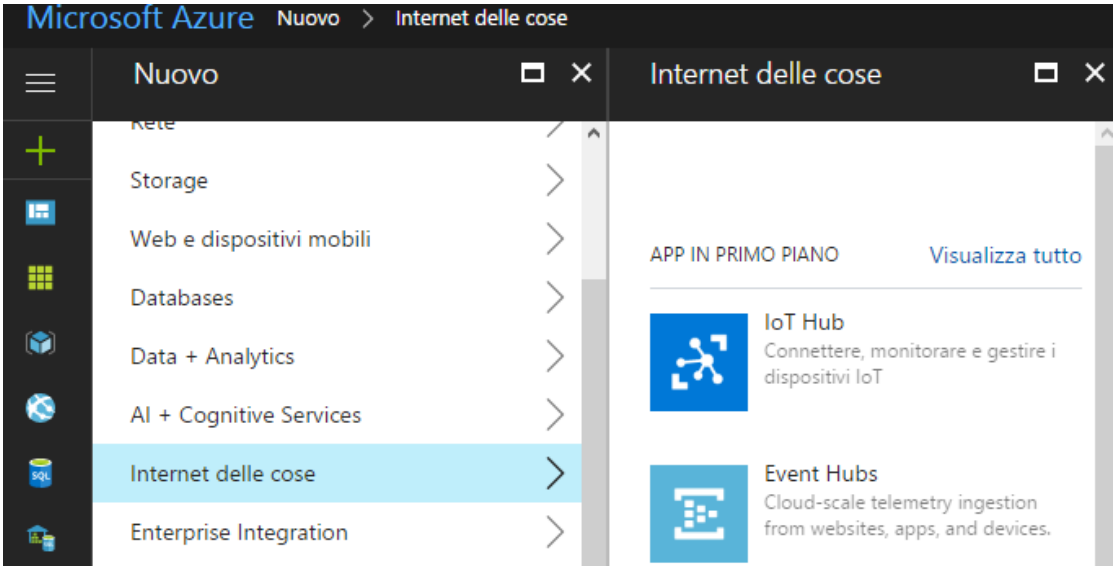


Fig.297

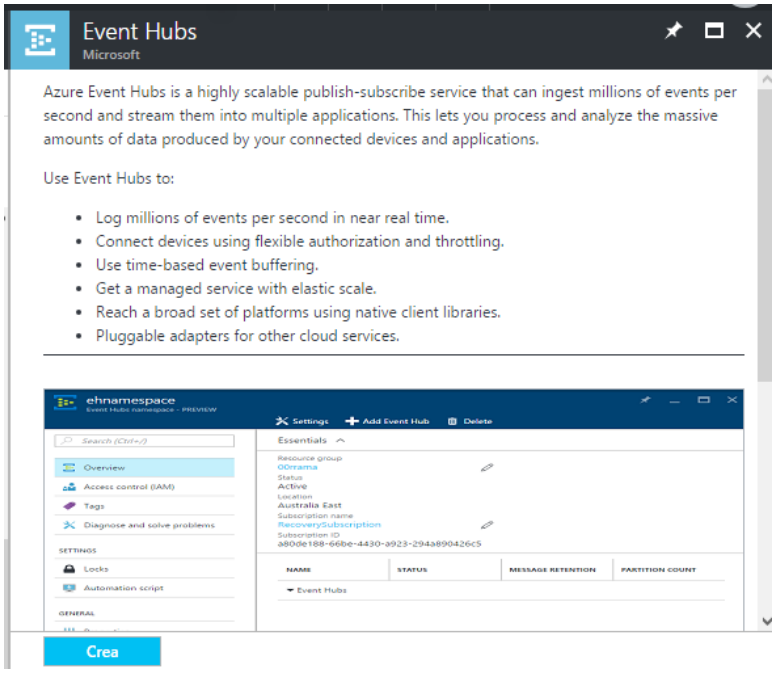


Fig.298

Premere il pulsante "Crea" di Fig.298 per iniziare a configurare il servizio, il quale verrà chiamato "eventiTX20" come stabilito in precedenza e come si vede chiaramente in Fig.299. Lasciare il piano tariffario di default, così come la sottoscrizione gratuita. Creare un nuovo gruppo ad hoc per i soli eventi chiamato "risorseEventiTX20" e procedere alla creazione del servizio tramite il pulsante "Crea", lasciando sempre abilitata la voce "Aggiungi al dashboard". In Fig.300 il collegamento sulla dashboard.

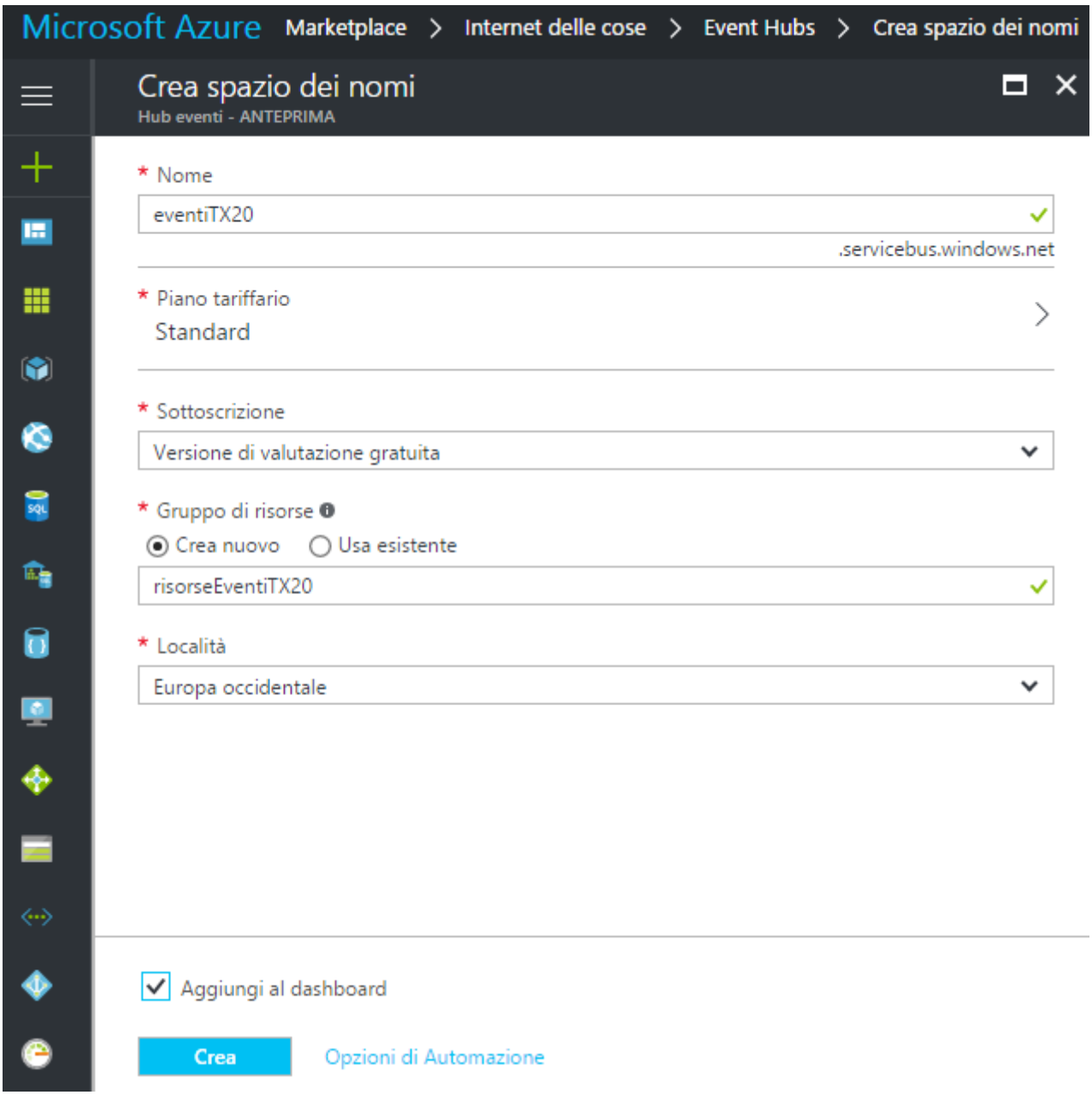


Fig.299

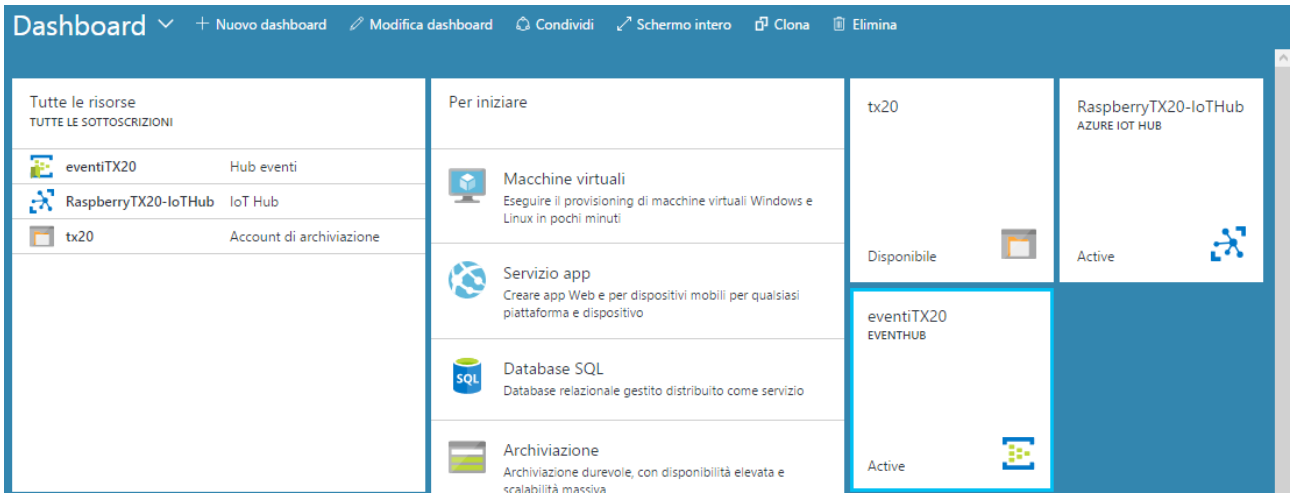


Fig.300

All'interno del servizio "IoT Hub eventiTX20" è necessario creare un gruppo che possa catalogare tutti gli eventi, quindi sulla falsa riga dei servizi precedenti, è necessario procedere alla creazione di un apposito gruppo selezionando il servizio stesso sulla dashboard come da Fig.301.

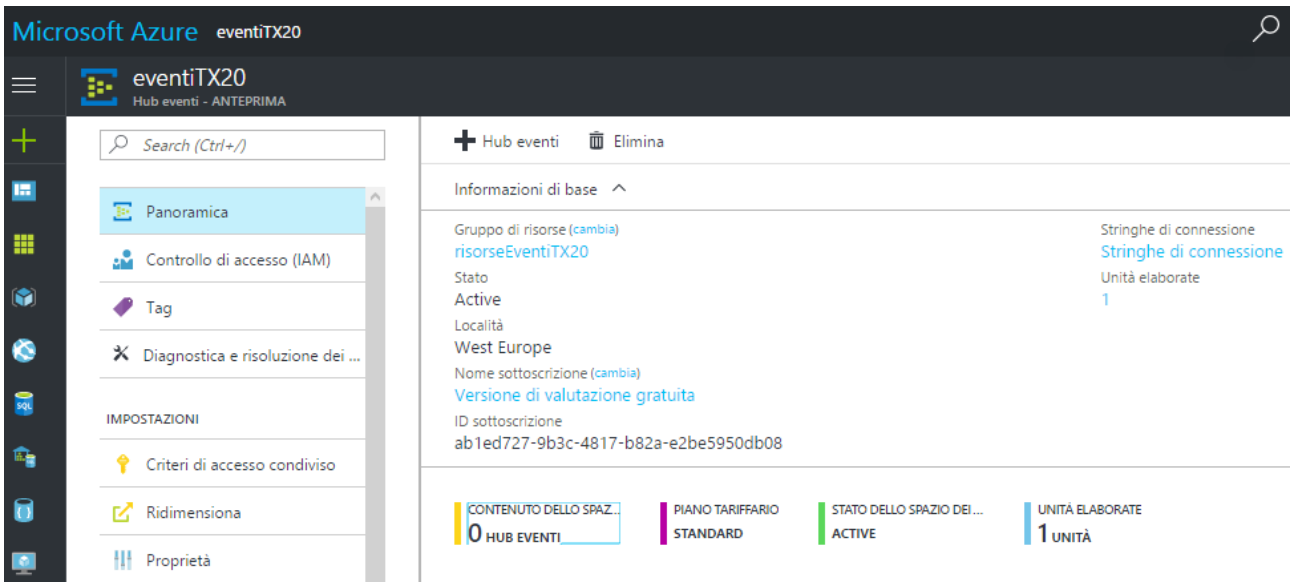


Fig.301

Successivamente selezionare, nella sezione "ENTITA'", la voce "Hub eventi" come da Fig.302 e installare uno specifico hub premendo il pulsante "+ Hub eventi". Il nome del hub è "hubTX20" come da Fig.303. Confermare tramite "Crea".

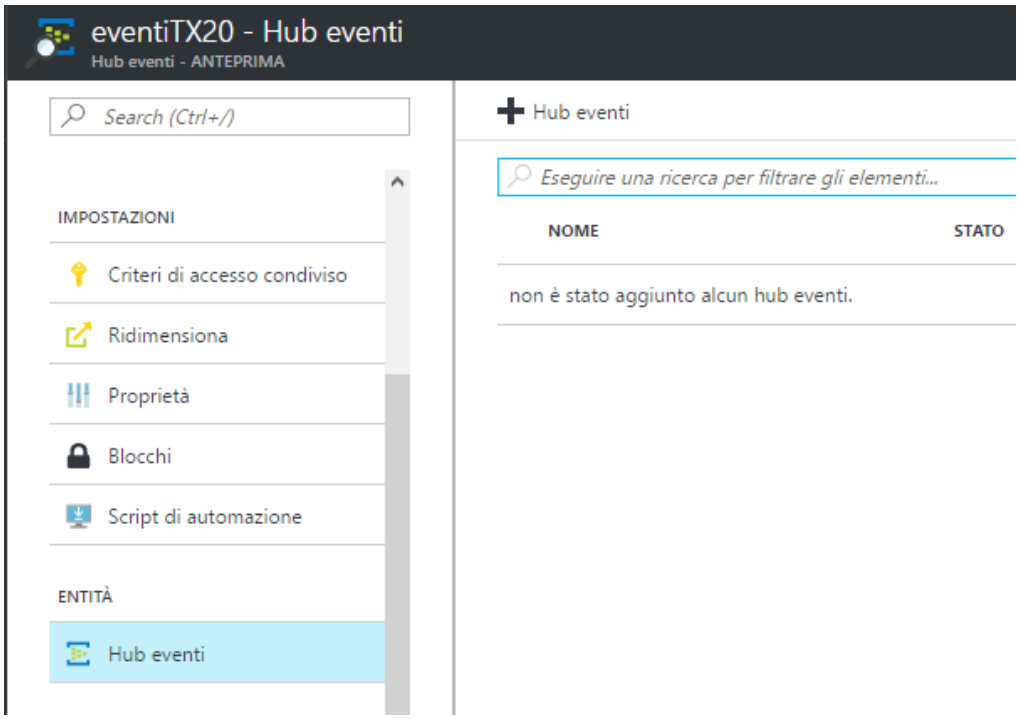


Fig.302

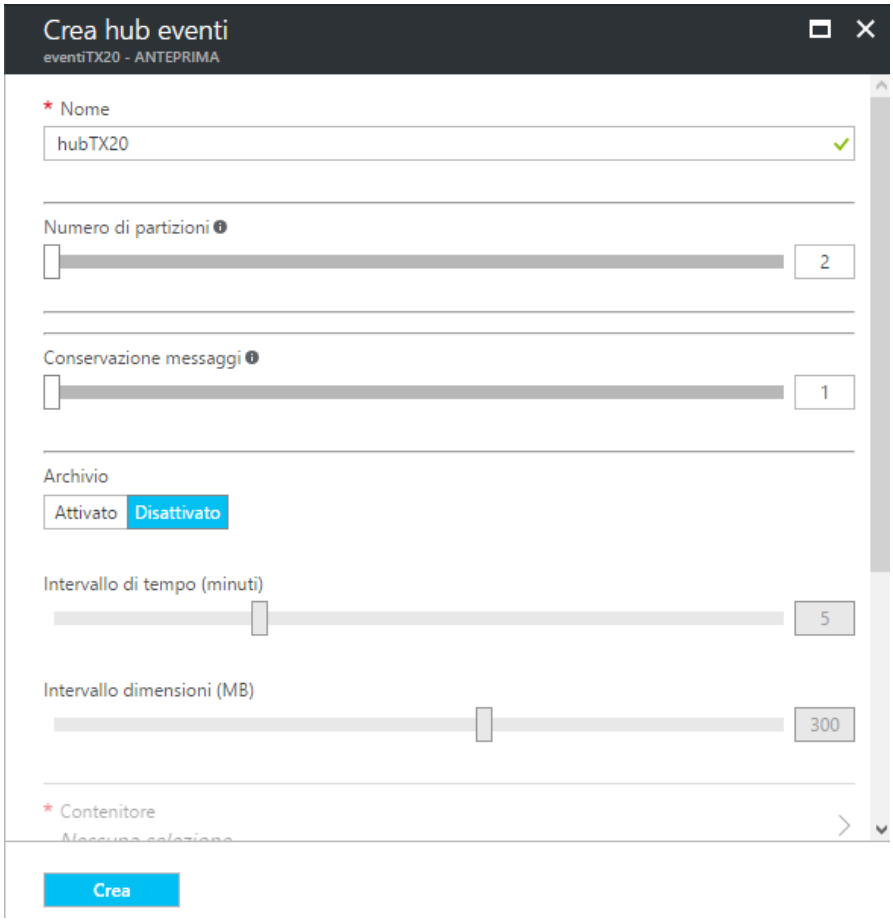


Fig.303

5.2.6 Stream Analytics

Questo servizio viene utilizzato come filtro sui dati che arrivano in ingresso sul cloud, grazie al servizio "IoT Hubs", così da potere decidere quali informazioni inviare nello storage BLOB e quali eliminare. In questo progetto il filtro non farà altro che copiare tutti i dati serializzati nel formato JSON direttamente nello storage, senza quindi bloccare nessun tipo di dato. Per aggiungere il servizio premere il pulsante "+ Nuovo" sulla dashboard e dal menù "Internet delle cose" selezionare la voce "Stream Analytics job" come da Fig.304. Procedere alla creazione tramite il pulsante "Crea" di Fig.305.

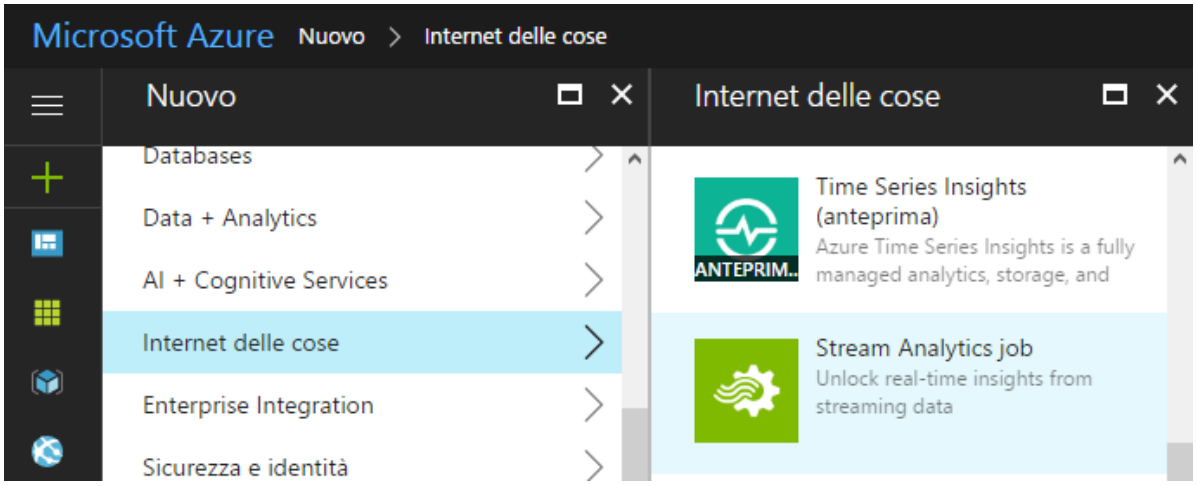


Fig.304

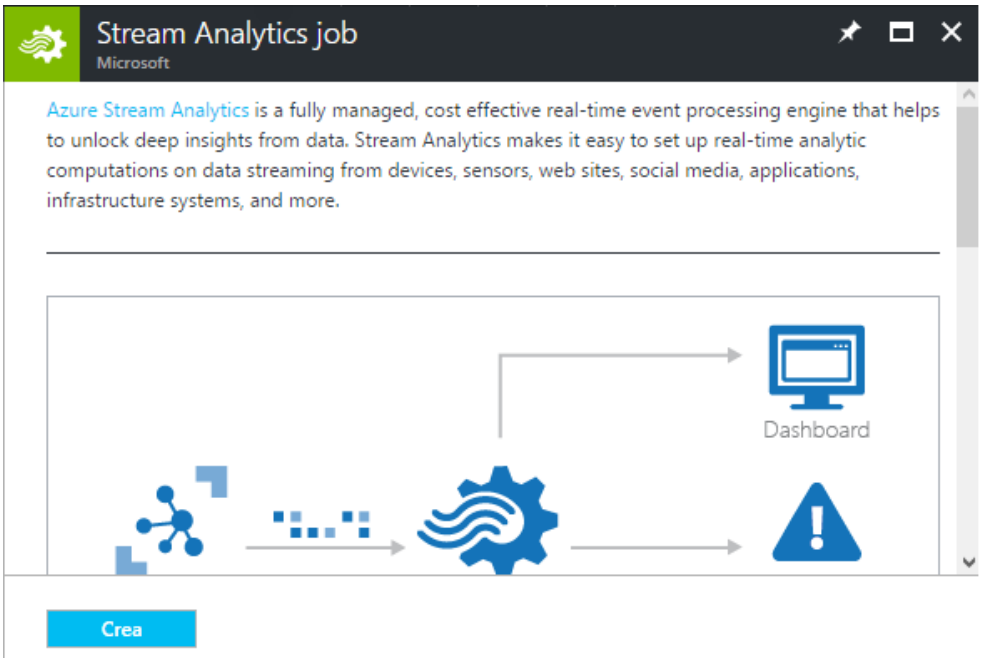


Fig.305

Il nome da assegnare al servizio è "streamTX20" e creando un nuovo gruppo di risorse ad hoc chiamato "risorseStreamTX20". Lasciando ovviamente la sottoscrizione gratuita procedere all'installazione del servizio con il solito collegamento sulla dashboard tramite il pulsante "Crea". Si vedano le Fig.306 e Fig.307.

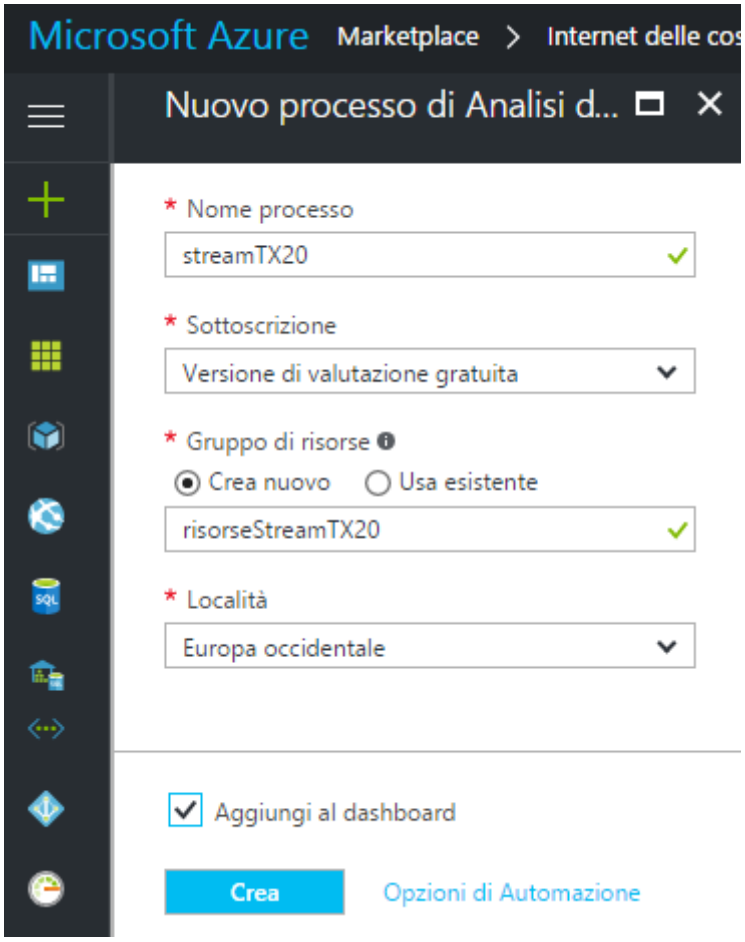


Fig.306

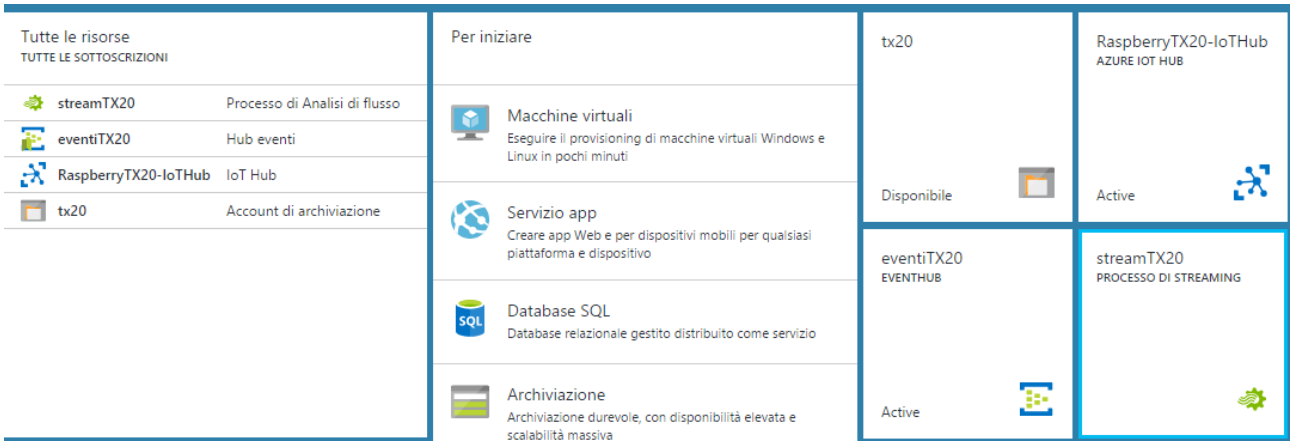


Fig.307

Selezionando il servizio appena collegato sulla dashboard, si vede che lo stream possiede due canali, uno di "Input" e l'altro di "Output". Il canale di "Input" riceve i dati dal servizio "IoT Hubs" e, tramite una "Query", è possibile decidere quali dati inviare sul canale di "Output" e, di conseguenza, allo storage BLOB. I due canali e la query vanno ovviamente configurati. Si veda la Fig.308.

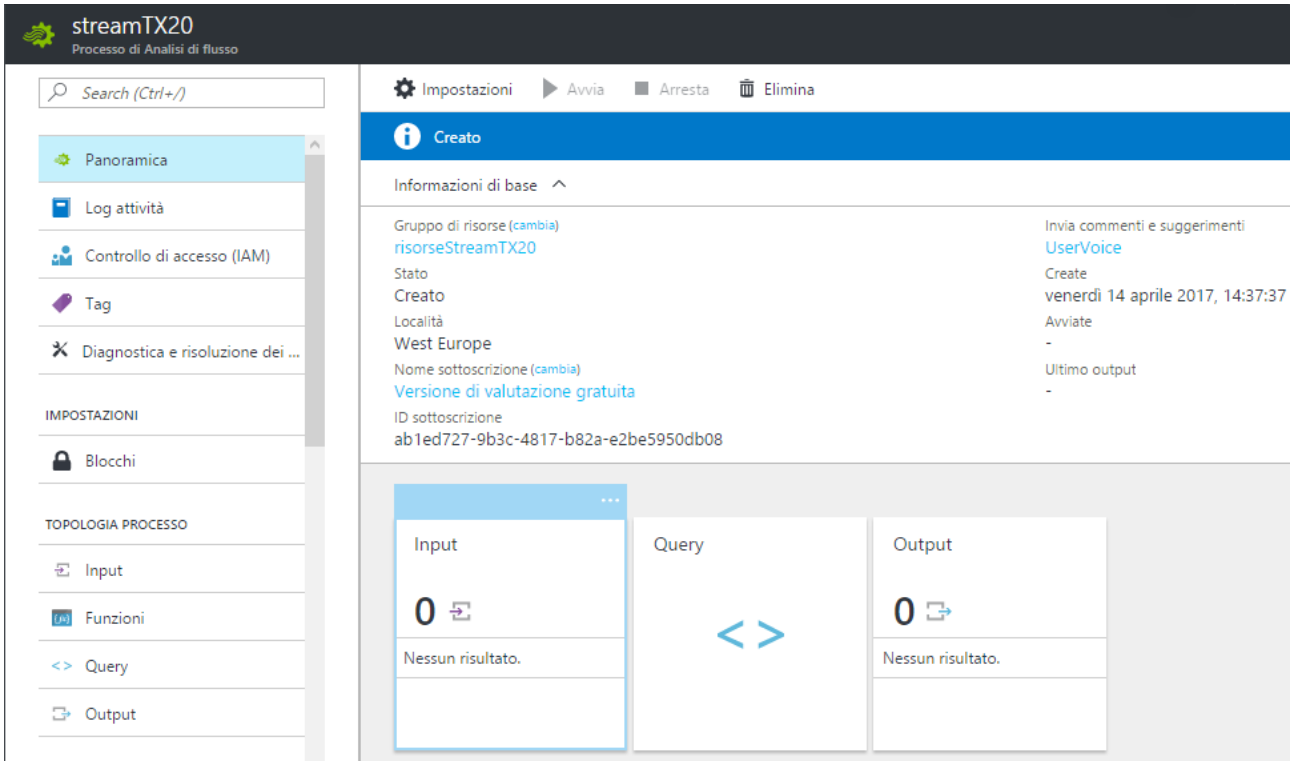


Fig.308

In Fig.309 si vede chiaramente che il servizio possiede 0 canali e 0 query.

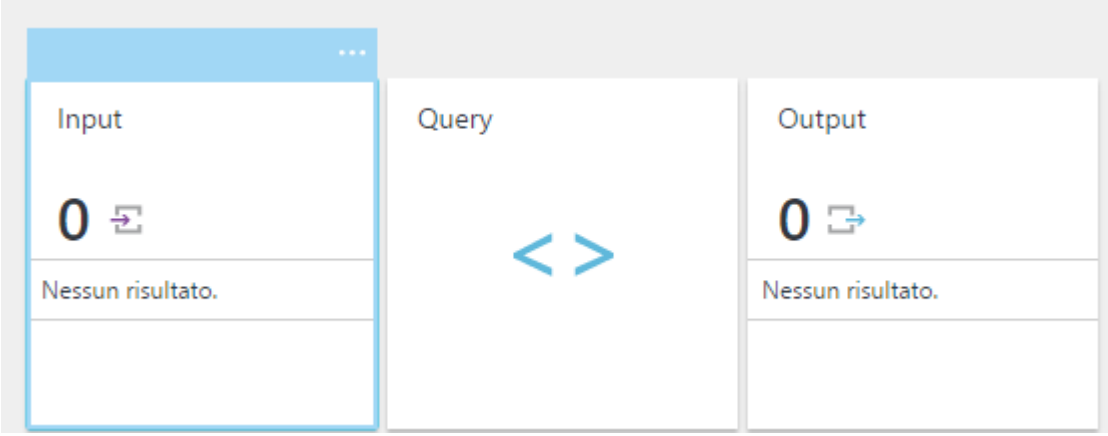


Fig.309

Cliccando sul riquadro "Input" della Fig.309, è possibile definire un nuovo input chiamandolo ad esempio "IngressoDelloStreamTX20" come si vede in Fig.310. Il tipo di dati è "Flusso dati" che proviene dall'origine "Hub IoT" utilizzando l'apposita sottoscrizione. Nella voce "Hub IoT" selezionare il nome del servizio precedentemente creato, ossia "RaspberryTX20-IoTHub". Il tipo di "Endpoint" è "Messaggistica". Verificare che il formato dei dati serializzati sia impostato su "JSON" e la codifica "UTF-8". Procedere alla creazione del canale di input tramite il pulsante "Crea".

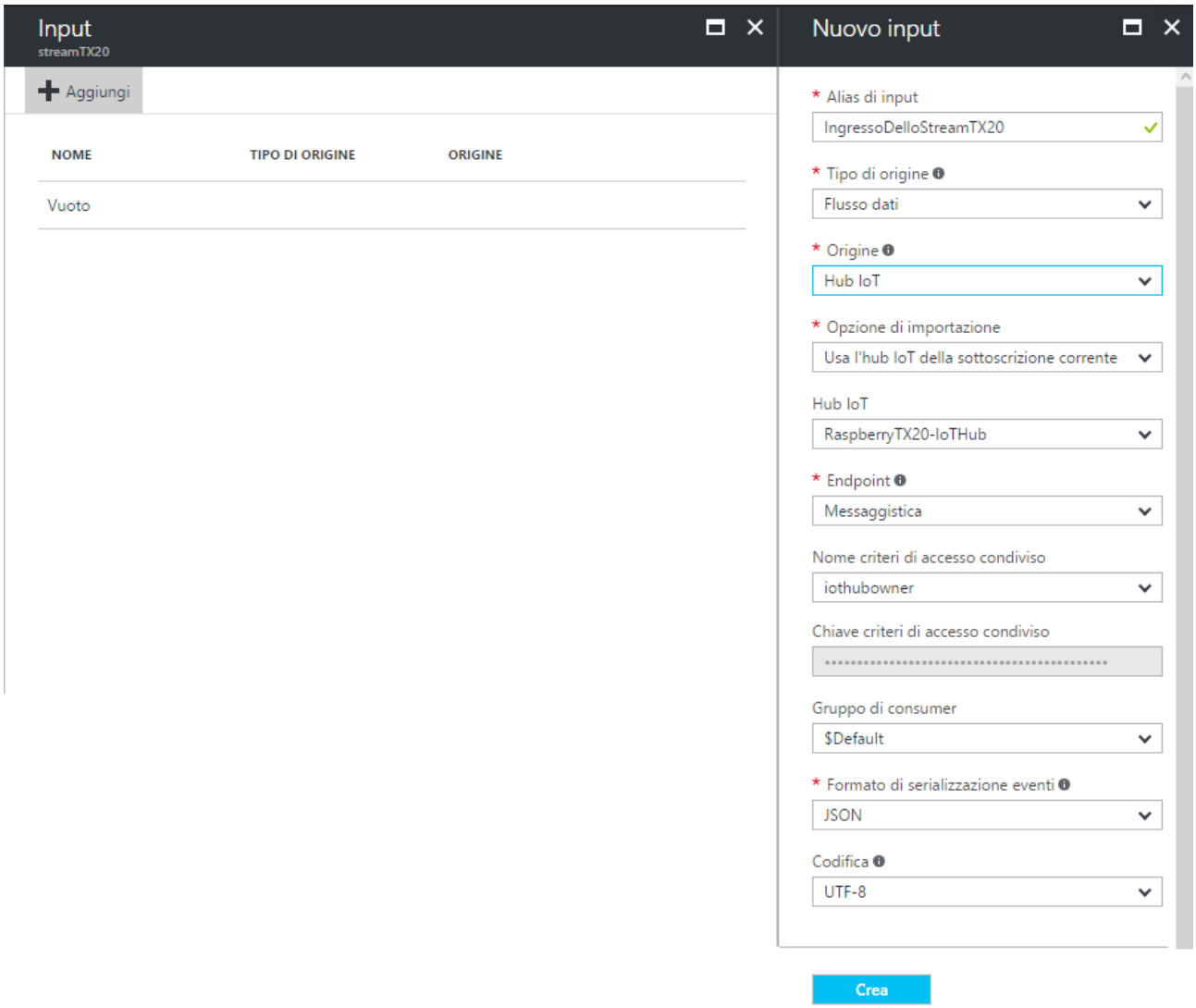


Fig.310

Il risultato della creazione del canale di input "IngressoDelloStreamTX20" è visibile in Fig.311.

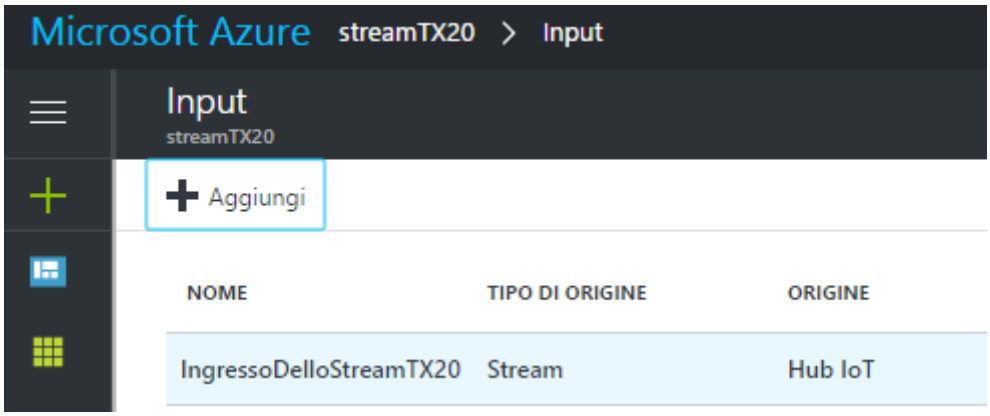


Fig.311

In Fig.312 si vede chiaramente la presenza di un canale in "Input".

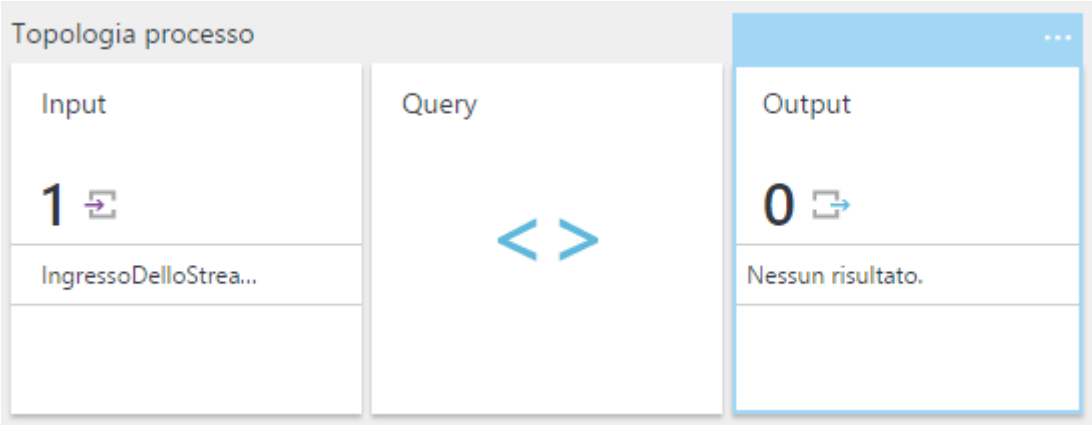


Fig.312

Terminata la configurazione dell'input, è necessario procedere alla creazione del canale di "Output", come da Fig.313. Il nome del canale è ad esempio "UscitaDalloStreamTX20", mentre nell'opzione "Sink" va indicata la tipologia dello storage, ossia "Archivio BLOB". L'account di archiviazione da utilizzare è "tx20", mentre il contenitore è quello creato nell'apposita sezione, vale a dire "dati". Il formato dei dati serializzati deve essere impostato su "JSON", mentre la codifica dei dati a "UTF-8". E' molto importante impostare il canale di uscita in modo da separare le informazioni ricevute dal canale di input, prima di inviarle allo storage. Il modo più semplice è avere un file JSON a righe, quindi selezionare nell'apposita listbox la voce "Separato da righe". Procedere alla creazione del canale premendo il pulsante "Crea".

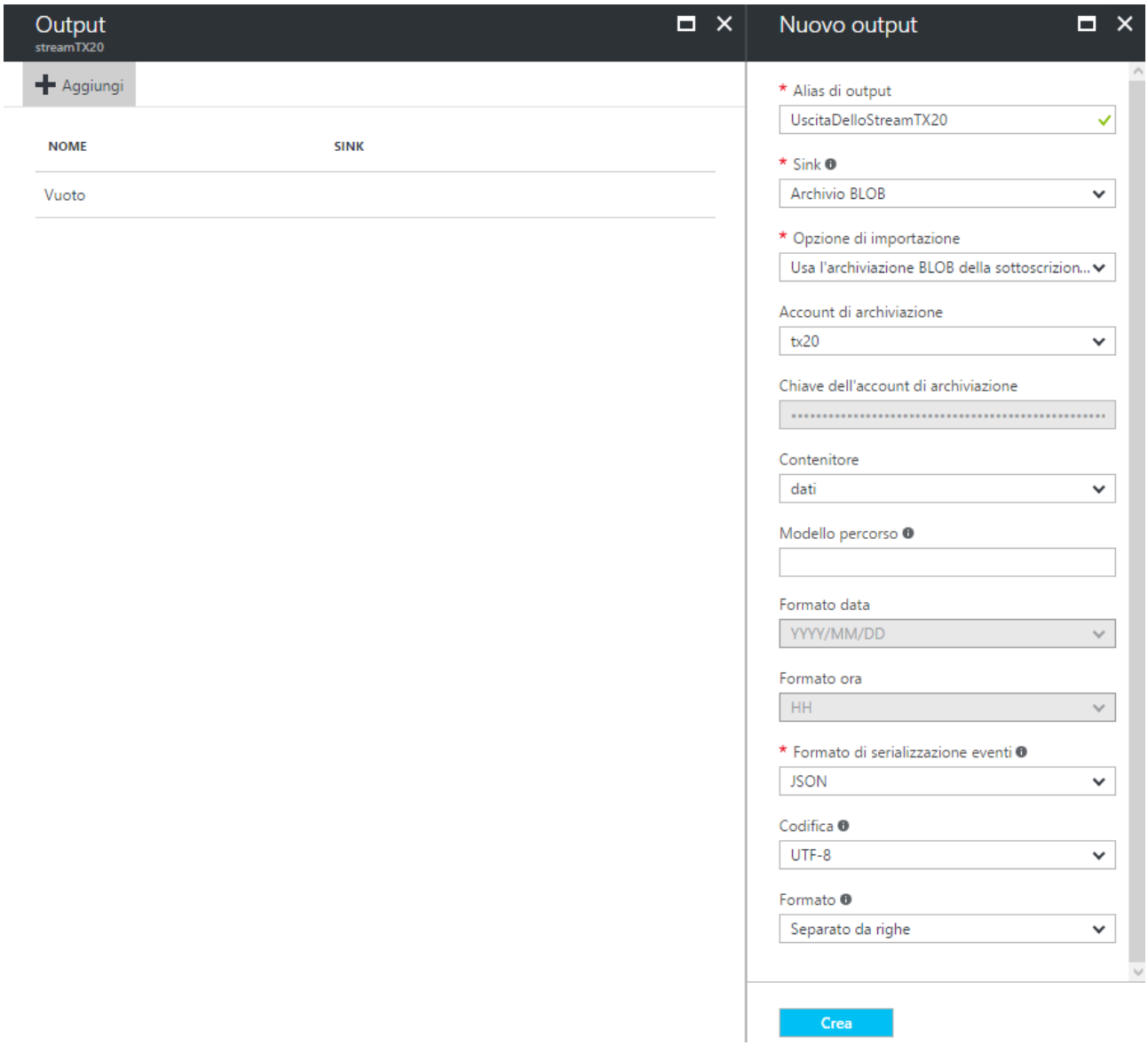


Fig.313

L'esito della creazione del canale è visibile in Fig.314.

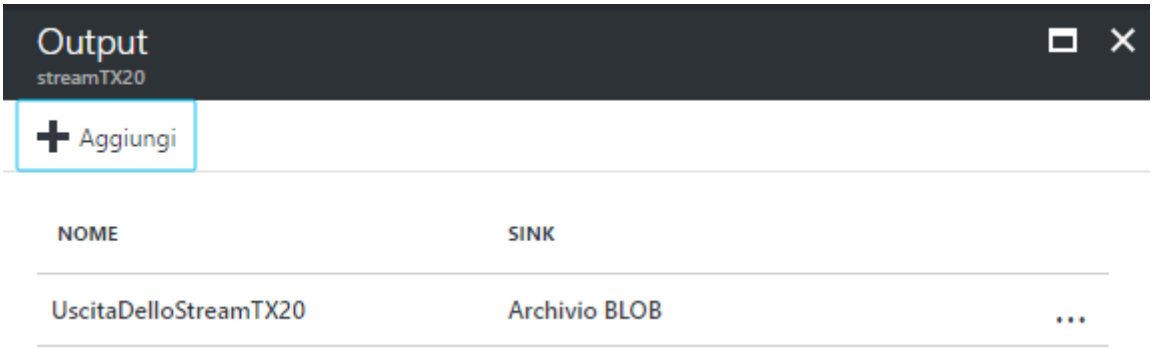


Fig.314

In Fig.315 si vede chiaramente la presenza di un canale in "Output".

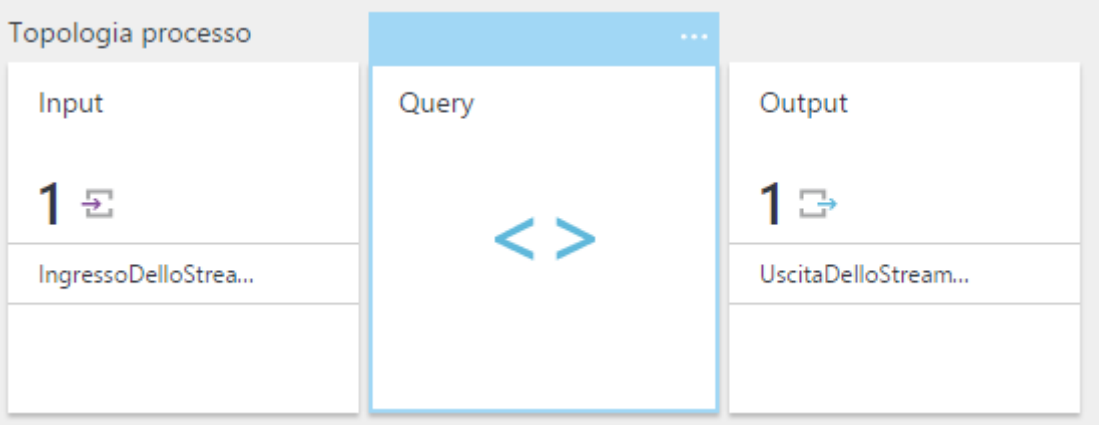


Fig.315

I due canali necessari al corretto funzionamento del servizio sono ora perfettamente impostati, manca solo la definizione del filtro che legga tutti i dati che arrivano sul canale di input e li inoltri su quello di output. Come già accennato in precedenza, in questo progetto non viene applicato nessun tipo di filtro personalizzato, ma si lascia passare tutto. L'impostazione del filtro prevede la creazione di una "Query", quindi è necessario selezionare la voce corrispondente in Fig.315.
La query ha una forma che ricalca il linguaggio SQL, infatti il passaggio dei dati da input ad output avviene tramite il seguente codice:

```
SELECT
    *
INTO
    [UscitaDelloStreamTX20]
FROM
    [IngressoDelloStreamTX20]
```

La prima clausola a venire processata, come avviene in SQL, è la FROM, ossia si seleziona l'origine dei dati, in tale contesto il canale di input "IngressoDelloStreamTX20". Successivamente la SELECT imposta la tipologia di filtro, che in questo progetto prevede la selezione di tutti i dati di input tramite il classico simbolo di '*'. Infine la INTO definisce la destinazione dei dati, vale a dire il canale di output "UscitaDelloStreamTX20".

In Fig.316 è riportata la query e come si vede è possibile, tramite il pulsante "Test", verificare il corretto funzionamento del codice scritto. L'esito del test è visibile in fondo alla figura nella sezione "Output generati:". La presenza di una sola riga è legata al fatto che il device IoT era stato programmato per inviare una ed una sola lettura del TX20, questo esclusivamente a scopo di test.

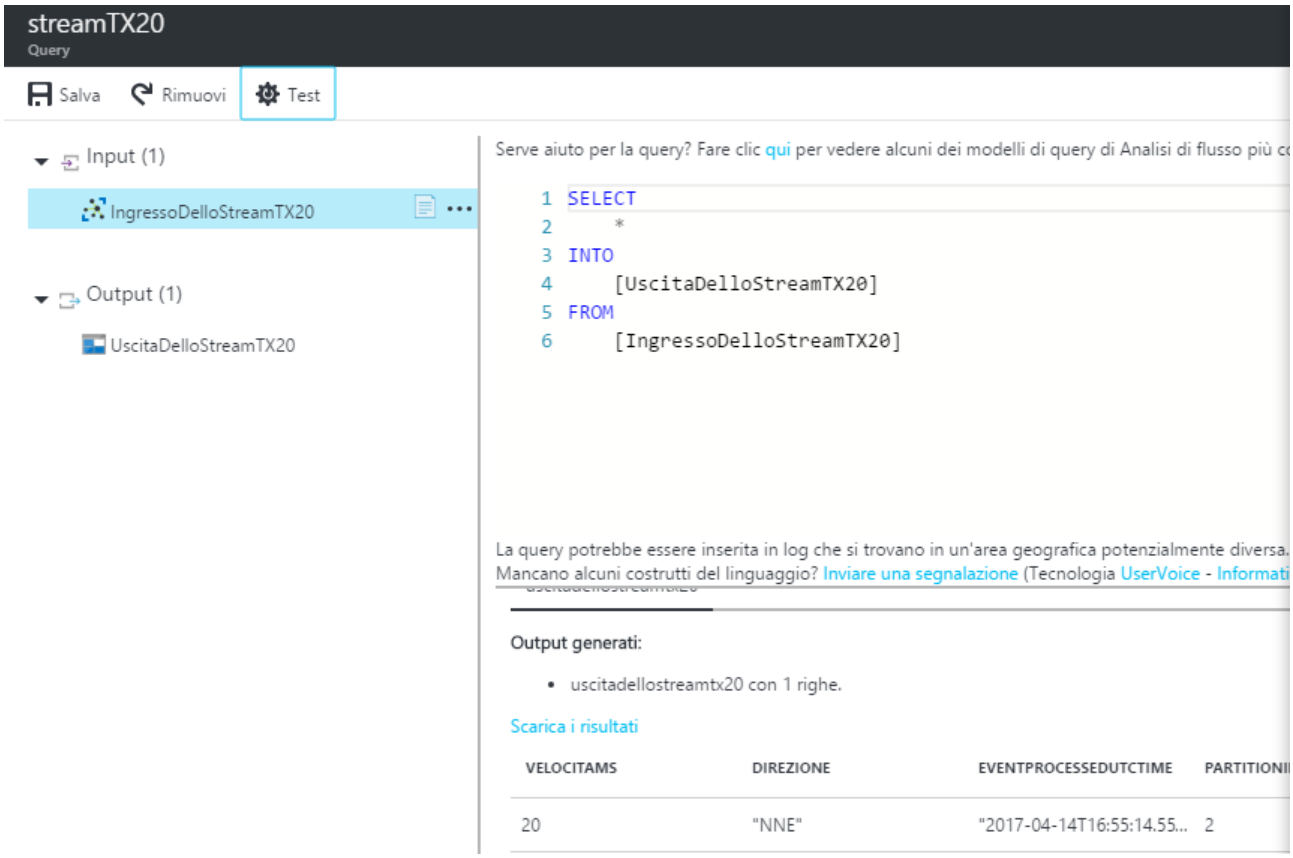


Fig.316

Terminata la query e verificato il corretto funzionamento, non resta che procedere al salvataggio del codice tramite il pulsante "Salva". Si veda la Fig.317.

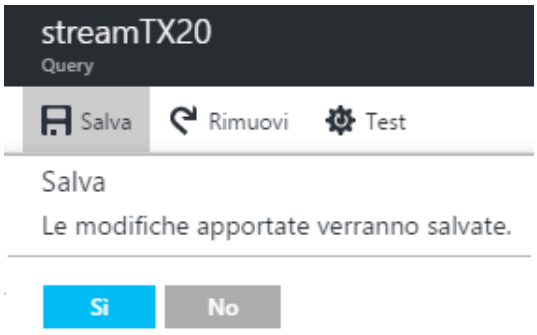


Fig.317

Il passo finale comporta l'attivazione del servizio di streaming "streamTX20" tramite il pulsante "Avvia", in caso contrario nessun dato verrà mai veicolato allo storage BLOB "dati", con l'ovvia impossibilità di qualsiasi tipo di visualizzazione web delle informazioni. La Fig.318 riporta questo semplice passo che spesso viene però dimenticato.

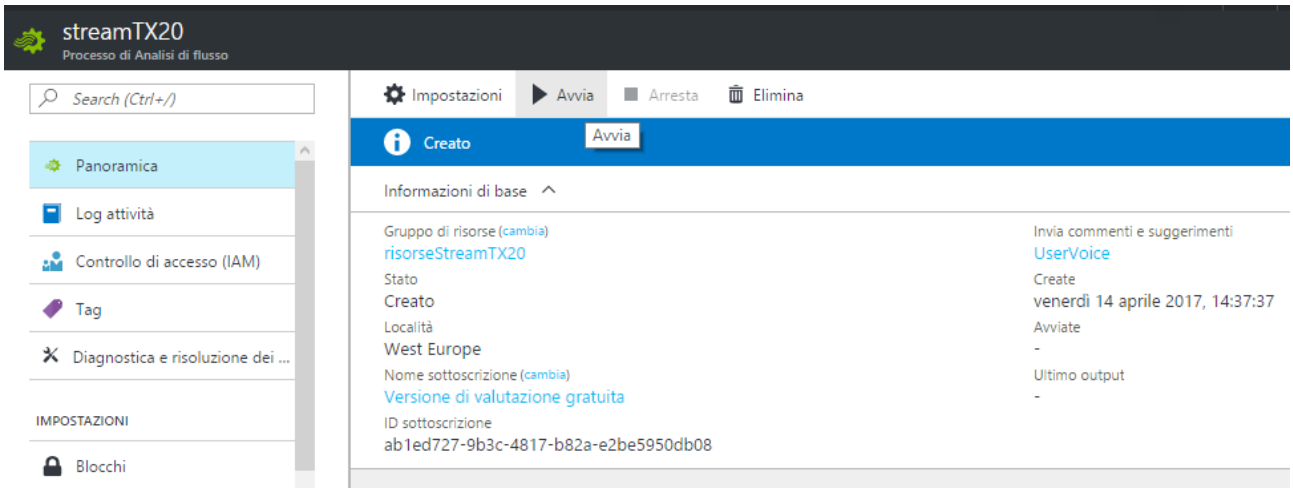


Fig.318

La richiesta di avvio del servizio avviene tramite conferma, come da Fig.319, premendo il pulsante "Avvia".

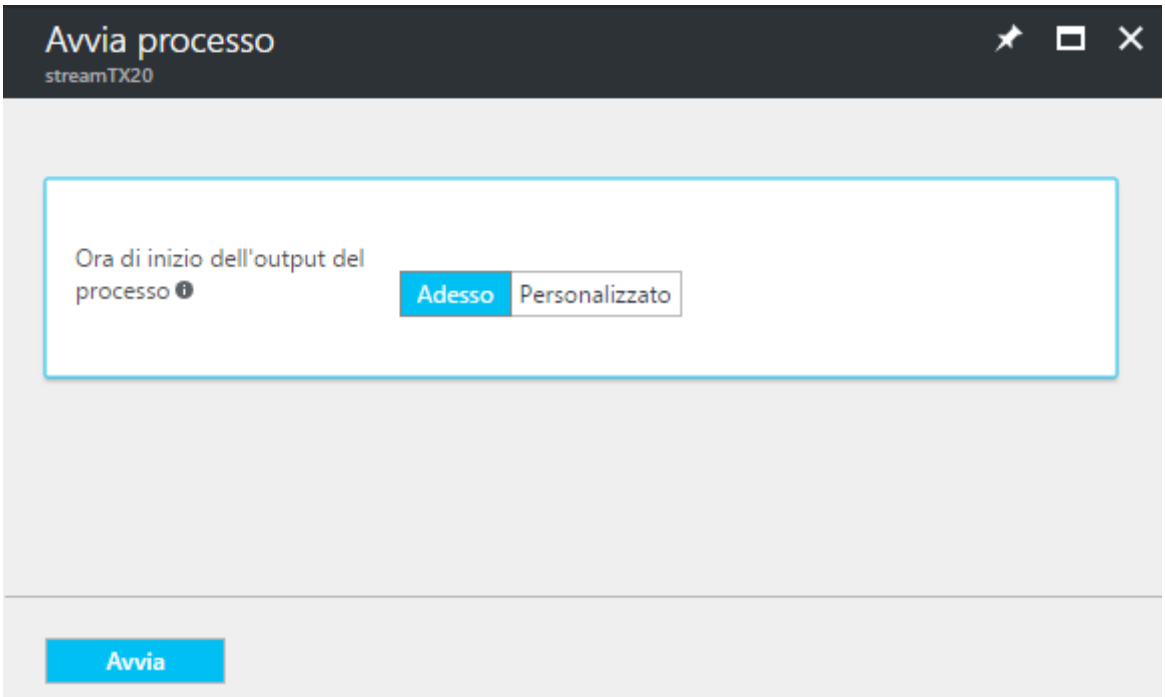


Fig.319

5.2.7 Sottoscrizione a pagamento

L'ultimo paragrafo di questa sezione affronta l'eventuale passaggio dalla sottoscrizione gratuita a quella con pagamento a traffico, così da completare questa breve discussione sulla configurazione di Azure. Dalla voce di menù "More services >" di Fig.320, selezionare "Sottoscrizioni" così da visualizzare il tipo di sottoscrizione utilizzata, vedesi la Fig.321.

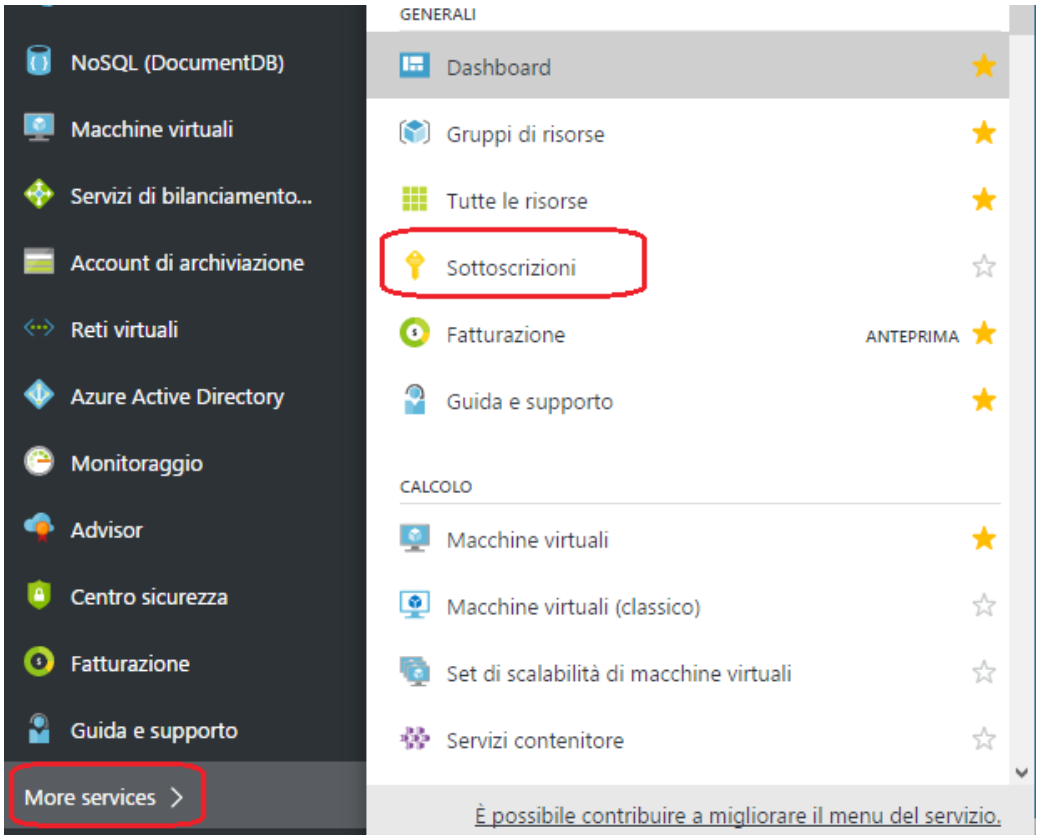


Fig.320

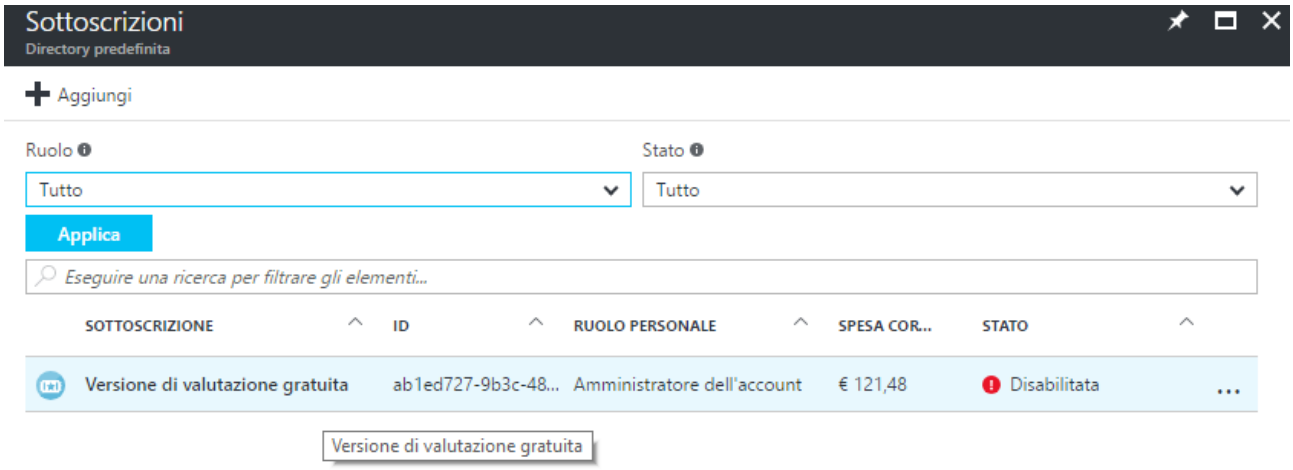


Fig.321

Premendo sulla dicitura "Versione di valutazione gratuita" è possibile verificare le spese relative al singolo servizio, come riporta la Fig.322.

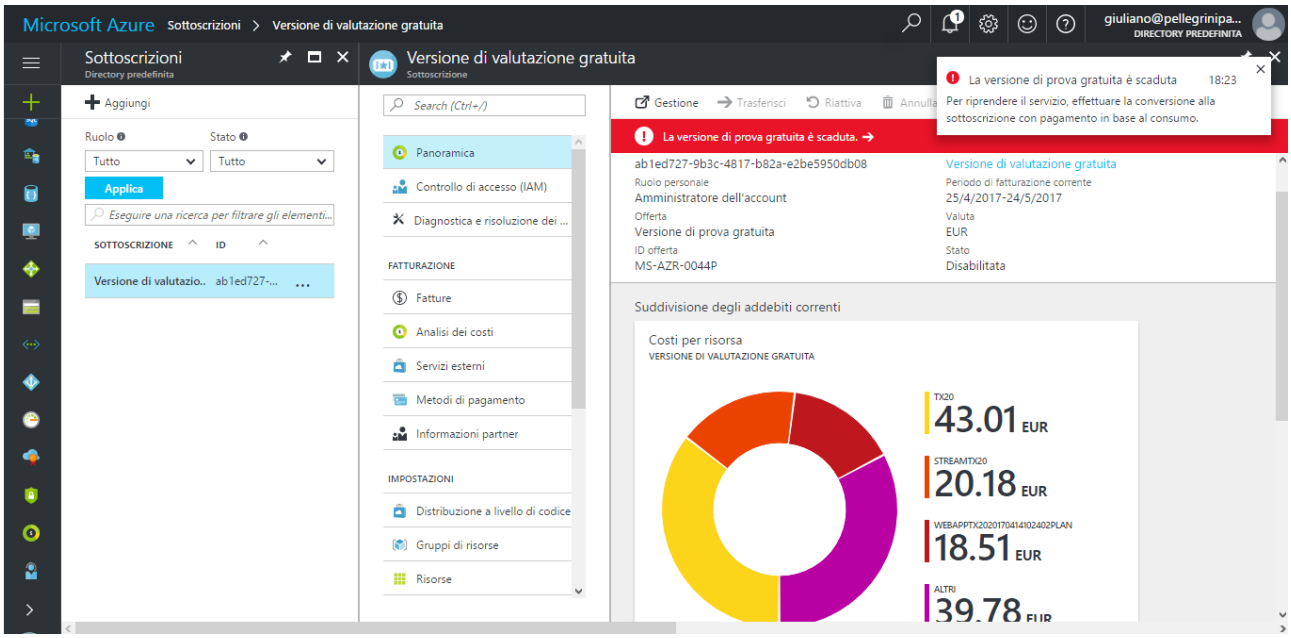


Fig.322

Cliccare sulla voce "La versione di prova gratuita è scaduta", come da Fig.323.

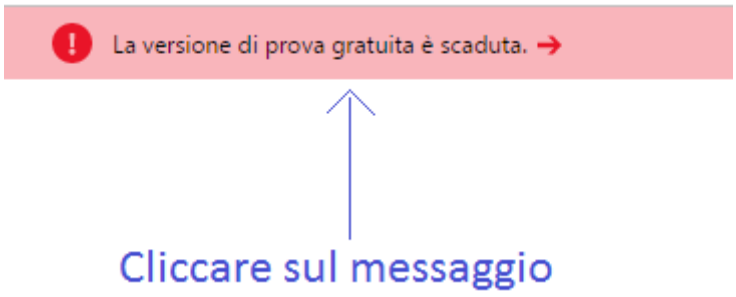


Fig.323

Come evidenziamo le immagini precedenti, la sottoscrizione è scaduta perché terminato il periodo di prova di 30 giorni. E' interessante notare che del credito iniziale gratuito di 170€, la configurazione dei servizi descritti in questo capitolo ed il traffico utilizzato, ai fini del test dell'applicazione, si ha avuto una spesa complessiva di circa 50€. Per ripristinare tutti i servizi è necessario convertire la sottoscrizione con una a pagamento, come indicato in Fig.324.

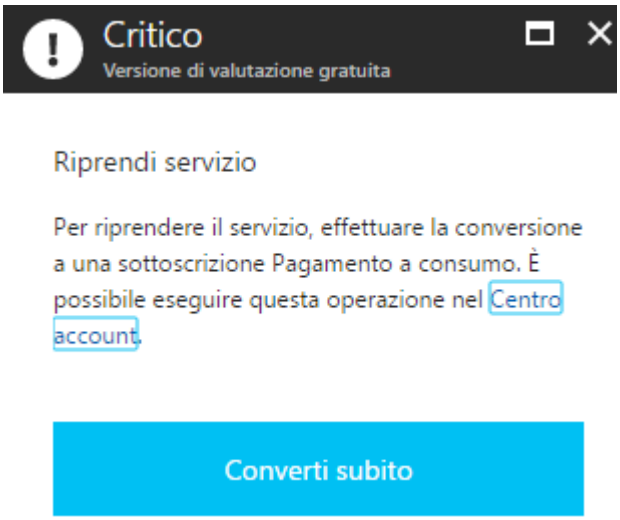


Fig.324

Una volta premuto il pulsante "Converti subito", viene presentato un riepilogo nel quale non è indicato nessun tipo di addebito, in fatti la cifra di addebito indicata è 0€, come si evince dalla Fig.325. Premere sulla sezione in ocra per convertire la sottoscrizione da gratuita a pagamento.

Fig.325

Selezionare la prima opzione di conversione, come da Fig.326, e poi procedere finalmente alla conversione premendo il pulsante "Aggiorna ora".

Fig.326

La sottoscrizione a pagamento è attiva con nessun addebito iniziale, come da Fig.327.

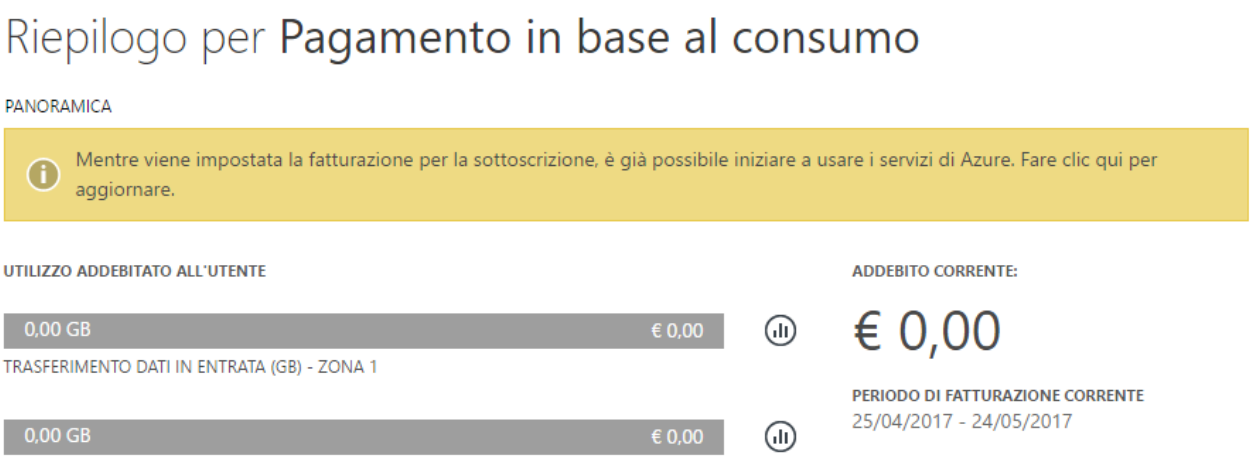


Fig.327

E' possibile visualizzare i dettagli del consumo tramite la voce "Scarica dettagli dell'utilizzo" come si vede in Fig.328.

Fig.328

Man mano che arriva traffico, è possibile in ogni momento vedere l'importo che verrà addebitato nella fattura mensile. Cliccare su "Pagamento in base al consumo" per graficare gli importi in base al servizio. In Fig.329 si vede che il servizio è ora attivo.

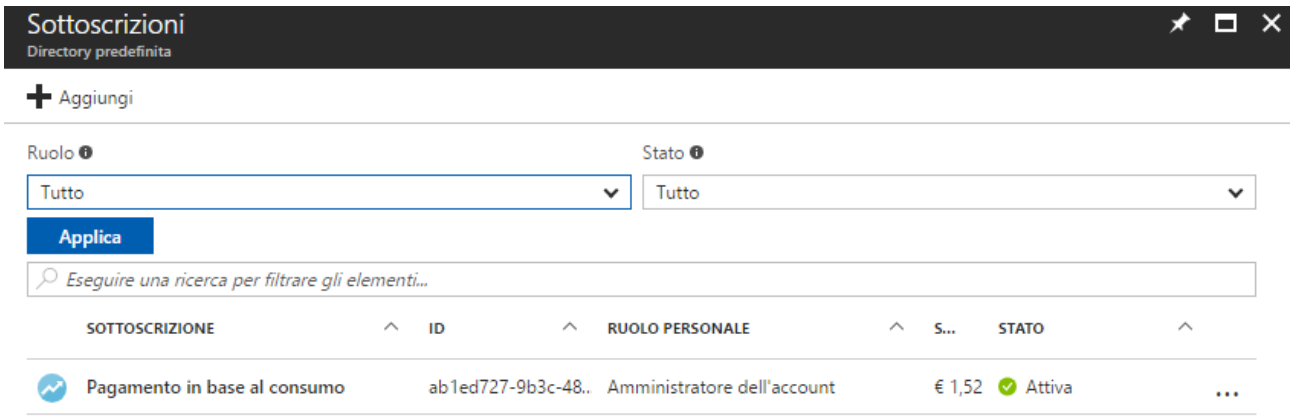


Fig.329

5.3 *Scrittura dati nel cloud*

La parte di configurazione del cloud Azure è fondamentale per avere la certezza che l'applicazione UWP possa inviare correttamente i dati al servizio "IoT Hubs". I dati letti dall'anemometro TX20 sono sempre la direzione e velocità del vento, nulla è cambiato, ma oltre che visualizzare le informazioni sul monitor connesso al Raspberry (se presente) e nella console di Debug, questi dati verranno anche inviati al cloud. In altri termini, tutto il codice descritto nel capitolo precedente resta invariato, ma sarà necessario scrivere le parti di connessione ad Azure, come era probabilmente intuibile.

5.3.1 Microsoft Azure Device Client

La connettività ad Azure è garantita da una serie di API messe a disposizione dalla Microsoft, librerie che sono facilmente scaricabili ed installabili direttamente dall'interno dell'ambiente di sviluppo Visual Studio 2015 o release successiva. Il primo step è quindi riaprire il progetto UWP C# del capitolo 4. Per effettuare l'installazione del pacchetto "Microsoft Azure Device Client", conviene appoggiarsi al packet manager integrato in Visual Studio chiamato "NuGet". Per utilizzare questa utility aprire il menù "Tools" e scegliere la versione grafica "Manager NuGet Packages for Solution..." di Fig.330.

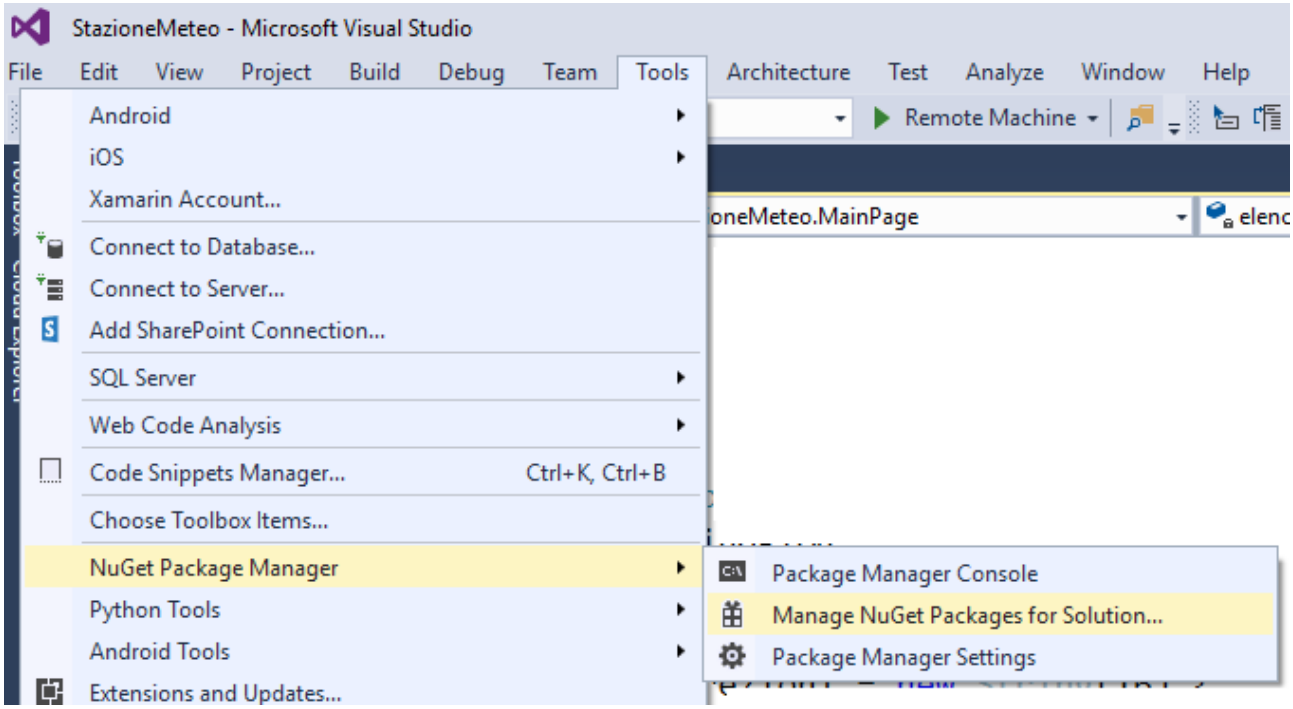


Fig.330

Nulla vieta di utilizzare la versione console e dare i comandi per l'installazione delle API, ma per comodità si preferisce la versione grafica. Nella sezione "Browse" inserire come chiave di ricerca "windows azure" e cercare il pacchetto "Windows.Azure.Devices.Client" versione "1.2.8" al momento della scrittura del seguente paragrafo. Selezionare nella parte di destra il progetto "StazioneMeteo" impostando però la release "1.2.1". Procedere infine all'installazione delle API nella soluzione tramite il pulsante "Install". La Fig.331 riassume questa fase.

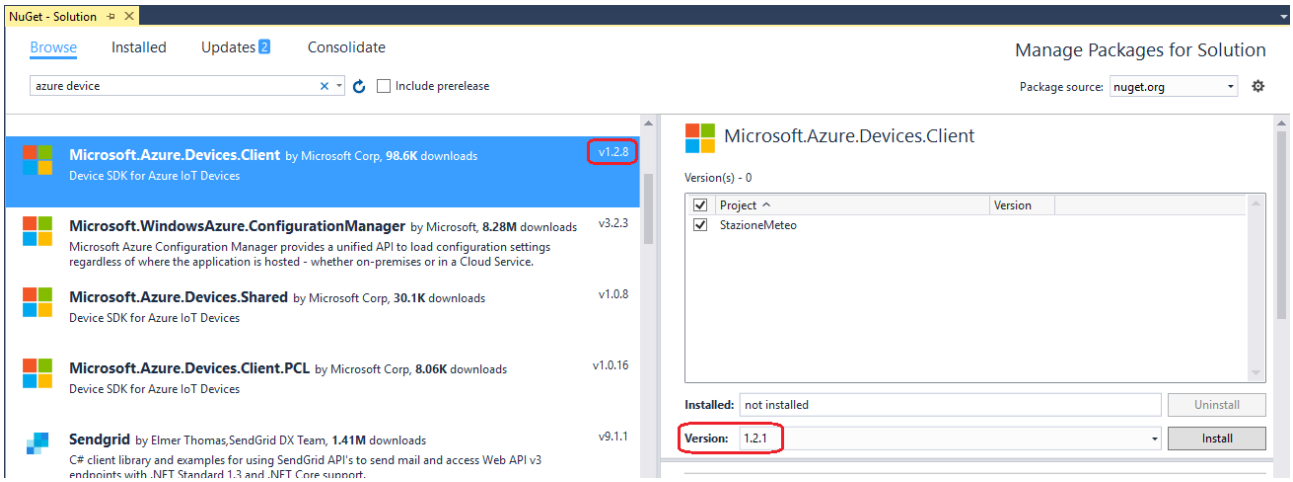


Fig.331

E' importante sottolineare che l'uso di queste API sono circoscritte esclusivamente al progetto "StazioneMeteo" visto che l'installazione avviene in tale soluzione. Nel momento in cui serva il pacchetto in altri progetti, andrà ripetuta l'installazione. Utilizzano questa logica è possibile installare varie release del pacchetto in progetti diversi. Ovviamente il peso del pacchetto di 110KB non è proprio insignificante, ma nemmeno gravoso in termini di spazio sul disco locale per questa applicazione. Il discorso potrebbe cambiare radicalmente in presenza di pacchetti oltre i 100MB o più, che comporterebbe un'inutile ridondanza con conseguente perdita di spazio. In simili situazioni è possibile pensare ad una specie di "installazione globale" del pacchetto, fatta una ed una sola volta e poi riutilizzato in tutti i progetti futuri. Questo approccio è poco usato e comunque vincola sulla versione installata e su eventuali aggiornamenti, quindi è una scelta che va ragionata in anticipo. L'installazione richiede altre librerie necessarie al pacchetto, come da Fig.332.

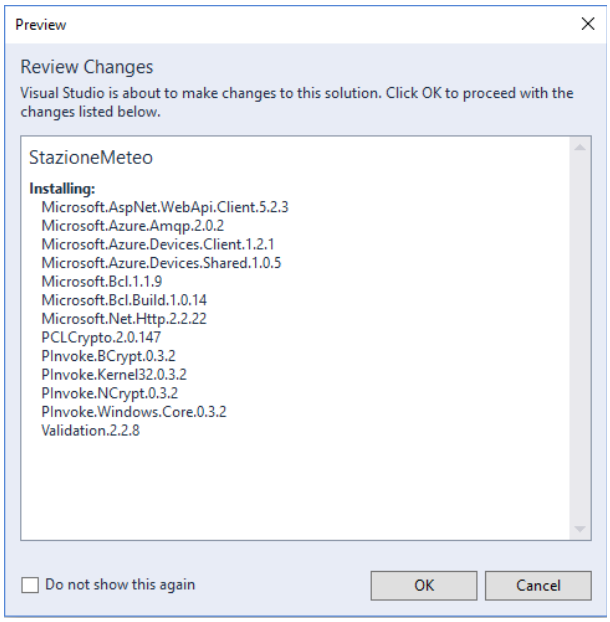


Fig.332

Procedere all'installazione delle librerie necessarie a "Windows.Azure.Devices.Client" premendo "OK" e accettando in seguito i termini di licenza "I Accept" di Fig.333.

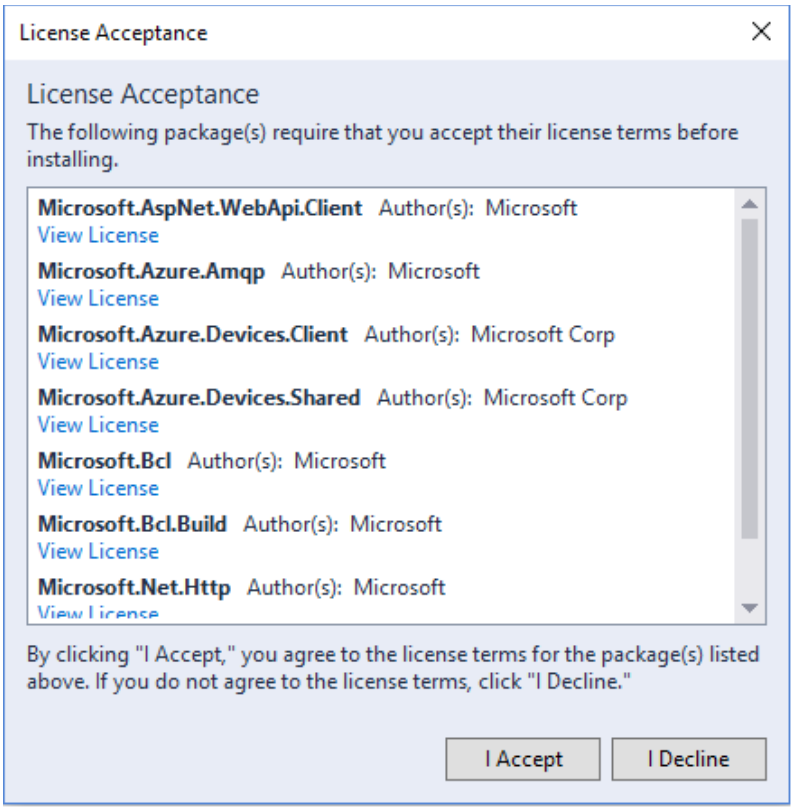


Fig.333

L'esito corretto dell'installazione inserisce nella sezione "References" della "Solution Explorer" la libreria "Microsoft.Azure.Devices.Client", necessaria per avere connettività con il cloud. Si veda la Fig.334.

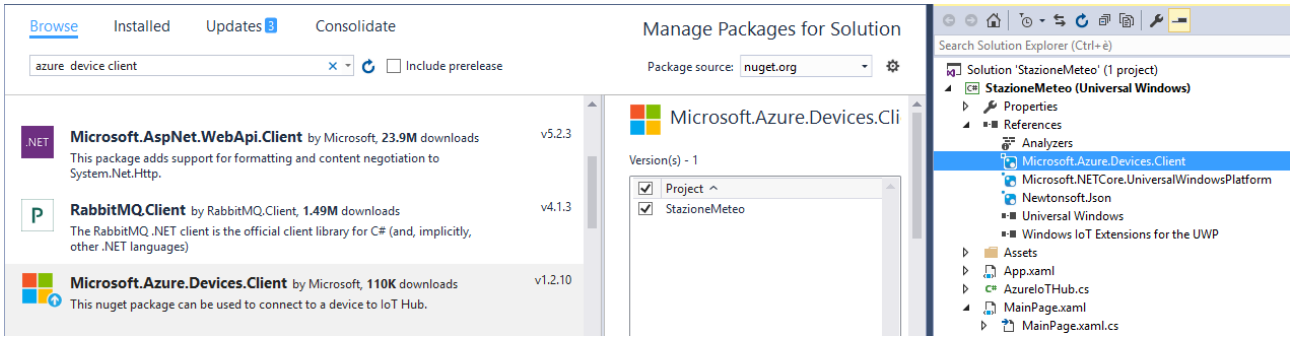


Fig.334

L'icona con la freccia bianca su sfondo blu presente sul nome del pacchetto nella sezione "Browse", indica che è presente una versione più aggiornata.

5.3.2 JSON.NET

La presenza del pacchetto per la connessione ad Azure deve essere accompagnata da una strategia che determini in che modo veicolare i dati verso il cloud. La scelta più consona per lo scambio dei dati da un qualsiasi device IoT verso il cloud, consiste nella rappresentazione delle informazioni in formato JSON "Javascript Object Notation", tecnica che oggi giorno è ampiamente diffusa in tutti i sistemi client/server. Le informazioni da veicolare nello "IoT Hubs" sono la direzione e velocità del vento, oltre che il giorno e l'ora del rilevamento dei dati. Queste informazioni a livello di applicazione UWP vengono inserite all'interno di un oggetto che verrà chiamato "TX20" nella memoria HEAP, di conseguenza è obbligatorio impostare un meccanismo che "trasformi" i bit dell'oggetto, in uno stream JSON da inviare al cloud. Questo meccanismo prende il nome di serializzazione, chiamato molto spesso "marshalling", e deve produrre uno stream i cui dati siano in formato JSON. Sarebbe anche possibile sfruttare il formato XML, ma la dimensione di uno stream JSON è minore, a parità di dati da serializzare, rispetto ad uno stream XML. Esistono altre differenze sostanziali tra XML e JSON, ma che esulano da questa discussione. La Fig.335 riporta in modo schematico il meccanismo di "marshalling", come si nota la classe "TX20" viene istanziata nello HEAP così da creare un oggetto, il

tutto tramite la clausola "new". Questa struttura deve venire convertita in un file JSON da inviare al cloud, motivo per cui la serializzazione viene affidata ad un framework ad hoc, chiamato JSON.NET. I campi della classe "TX20" sono "VelocitaMS", "Direzione", "Data" e "Ora" e verranno riportati in modo identico anche nel file JSON con i rispettivi dati. Ogni blocco di informazione riguardante la singola lettura è racchiuso con delle graffe dentro il file JSON. Nella Fig.335 a titolo di esempio sono state riportate due letture.

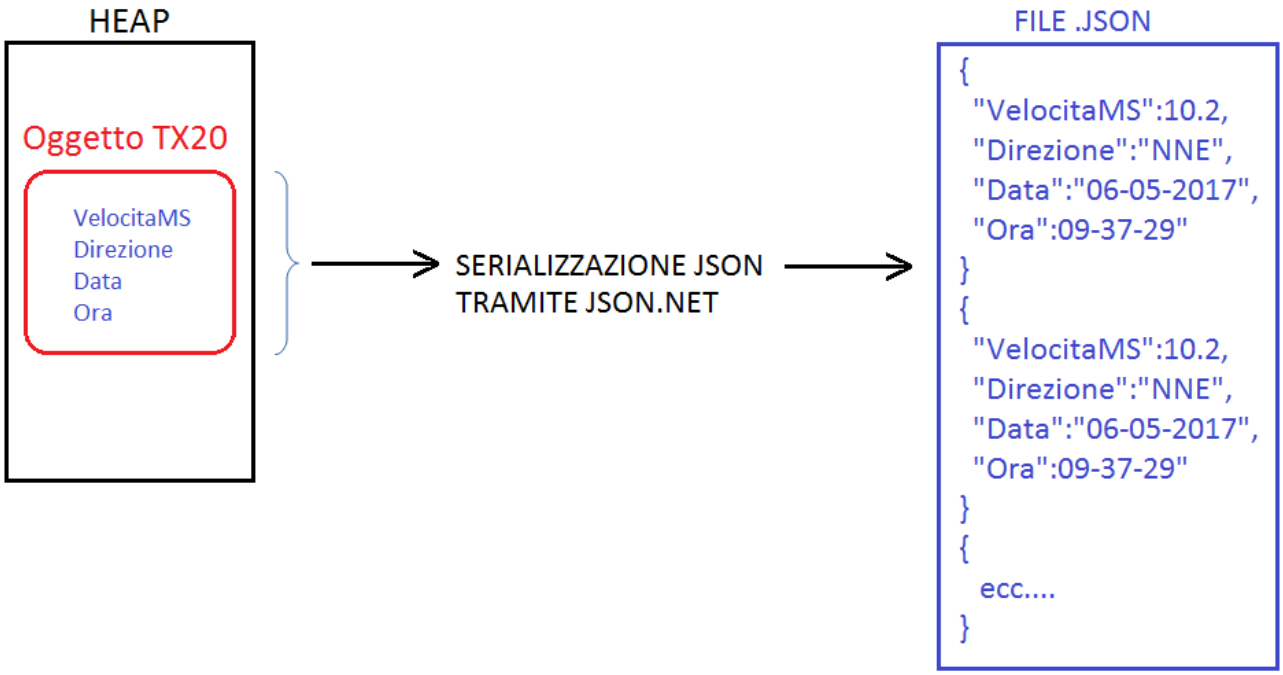


Fig.335

Il nome del pacchetto che rappresenta JSON.NET è "Newtonsoft.Json", pacchetto la cui versione "6.0.8" è già stata installata come requisito base per il precedente gruppo di librerie "Windows.Azure.Storage", come da Fig.332. Il packet manager molto probabilmente indicherà la presenza di una serie di "Updates", tra cui lo stesso "Newtonsoft.Json" versione "10.0.2" al momento della stesura di questo paragrafo. Quest'ultima release non è però compatibile con il .NET Core UWP 5.1.0 di Visual Studio 2015, ma serve quello della versione 2017, motivo per cui è più che sufficiente, per gli scopi di questo progetto, aggiornare JSON.NET alla versione "9.0.1". Questa problematica della compatibiltà con il .NET Core vale per tutti i pacchetti gestiti da "NuGet", quindi è necessario porre la massima attenzione. Aggiornare il pacchetto .NET Core UWP 5.1.0 alla versione "5.3.3" richiede Visual Studio 2017. Si veda la Fig.336.

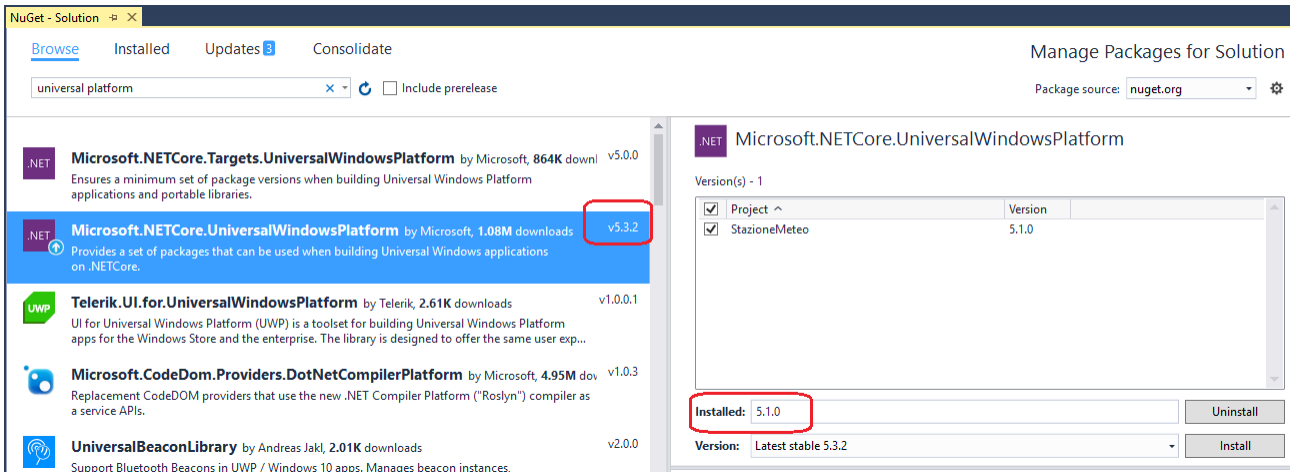


Fig.336

L'aggiornamento del pacchetto "Newtonsoft.Json 9.0.1" segue la medesima logica del pacchetto precedente, quindi è necessario selezionare la soluzione "StazioneMeteo" e premere "Install". Confermare poi l'update del pacchetto. Si vedano la Fig.337 e Fig.338.

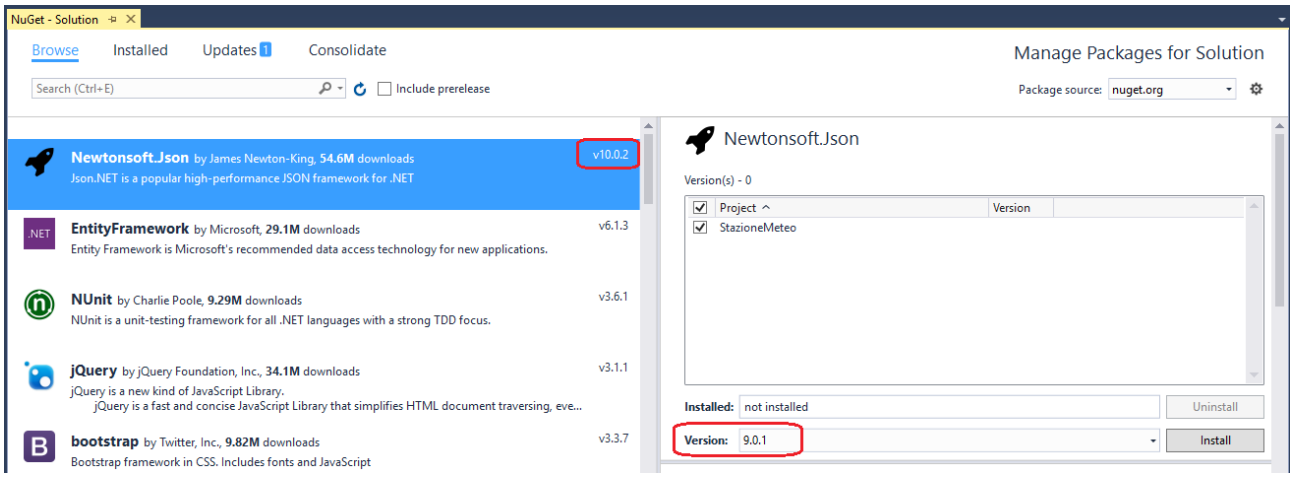


Fig.337

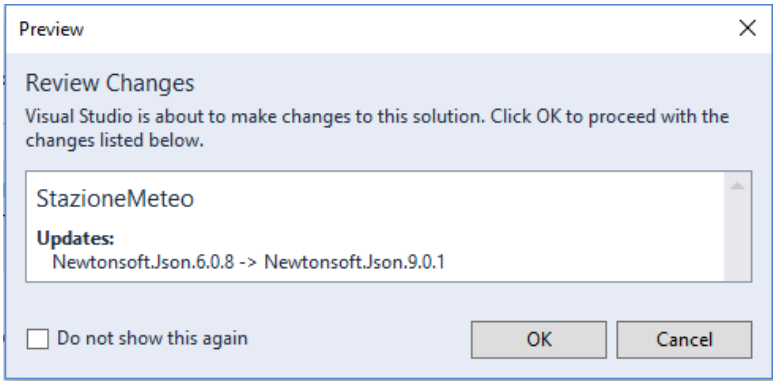


Fig.338

5.3.3 Progetto stazione meteo

L'inserimento delle librerie del paragrafo precedente sono un requisito essenziale per permettere alla UWP C# di inviare i dati al cloud serializzandoli in formato JSON. Rispetto al capitolo 4, è necessario aggiungere delle classi per una gestione corretta della serializzazione e del cloud. Le classi da creare da zero nel progetto sono:

- TX20
- TX20Exception
- AzureIoTHub

Procedere alla creazione di queste tre classi all'interno della soluzione "StazioneMeteo" come indicato in Fig.339.

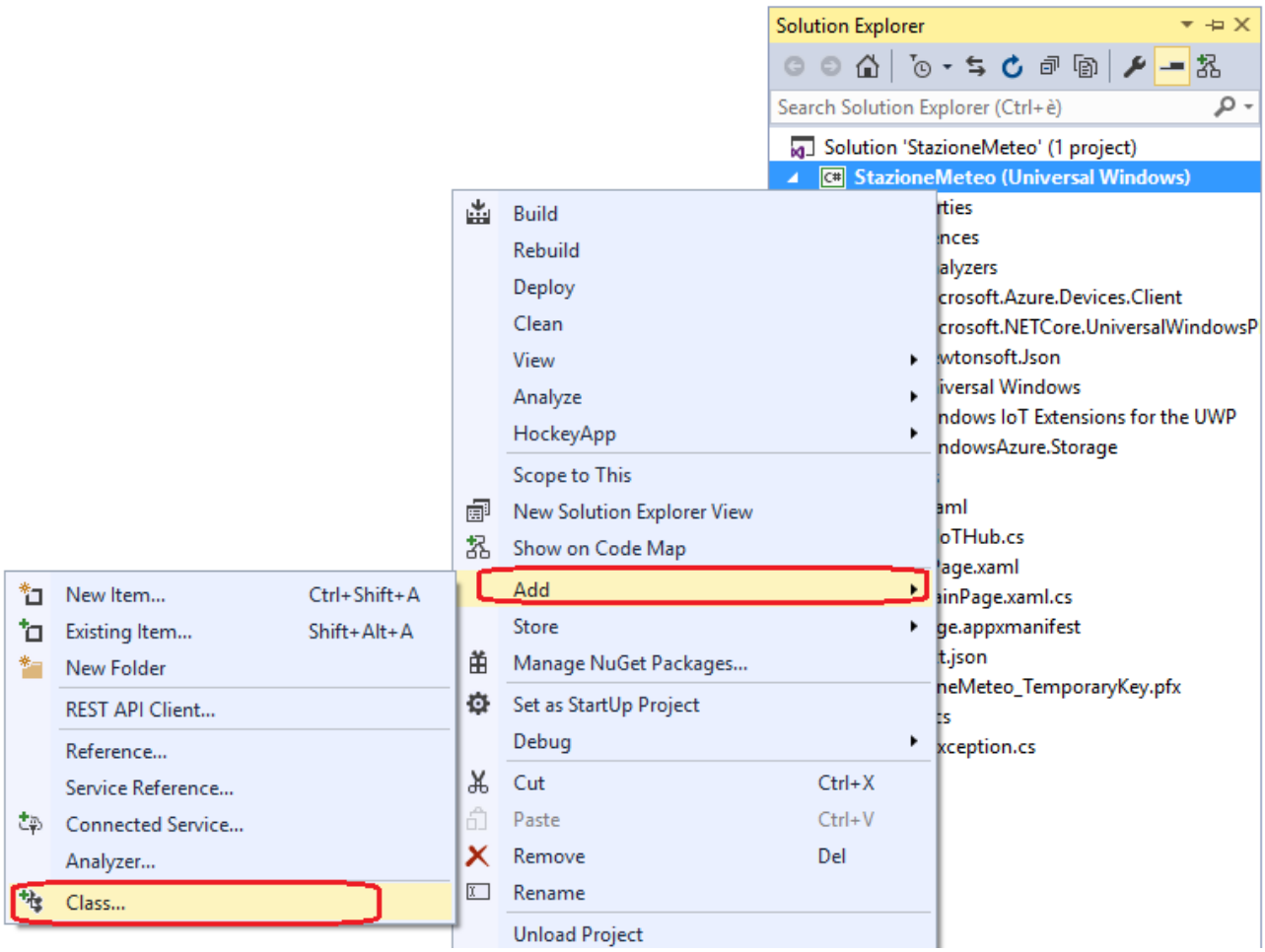


Fig.339

Le Fig.340, Fig.341 e Fig.342 mostrano come creare le classi. Premere "Add".

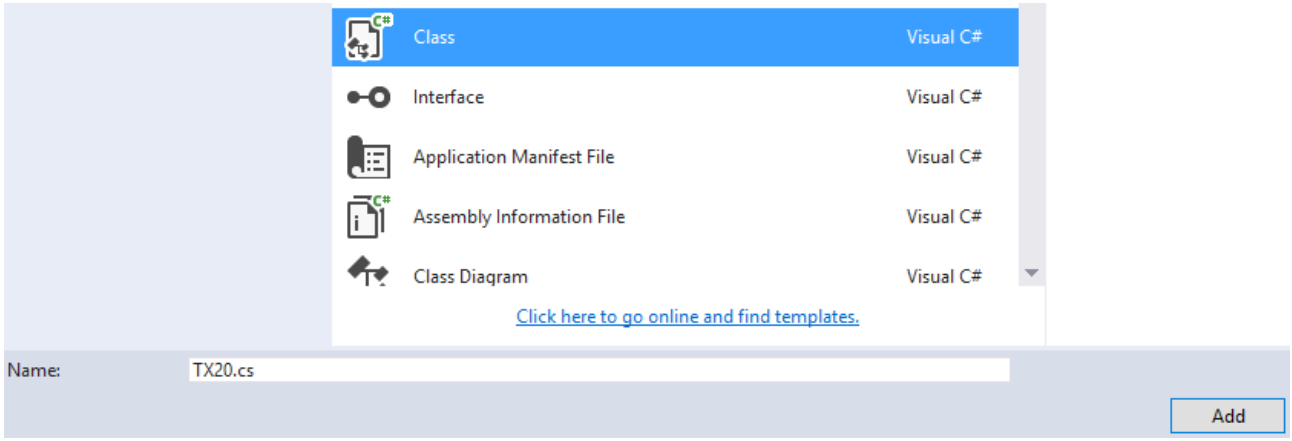


Fig.340

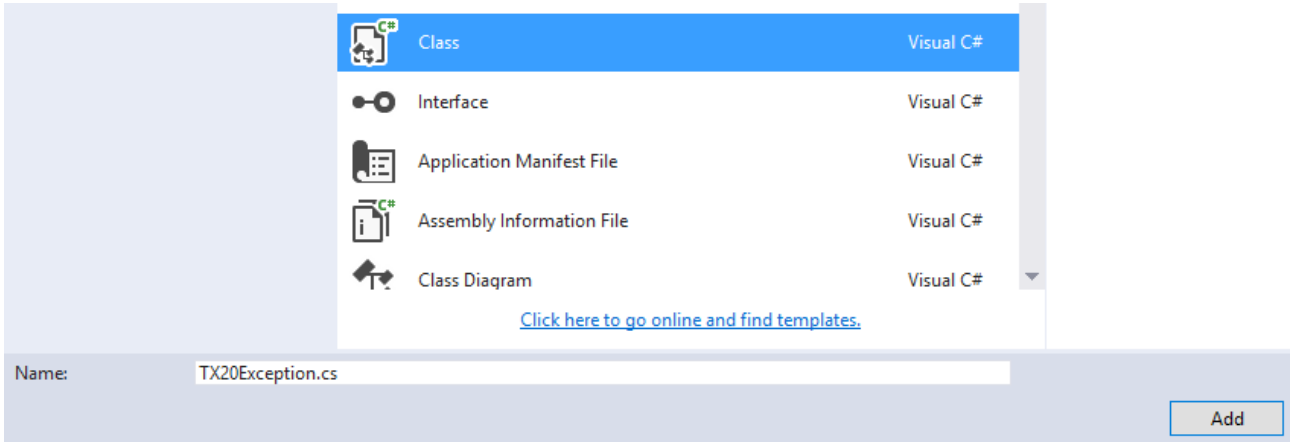


Fig.341

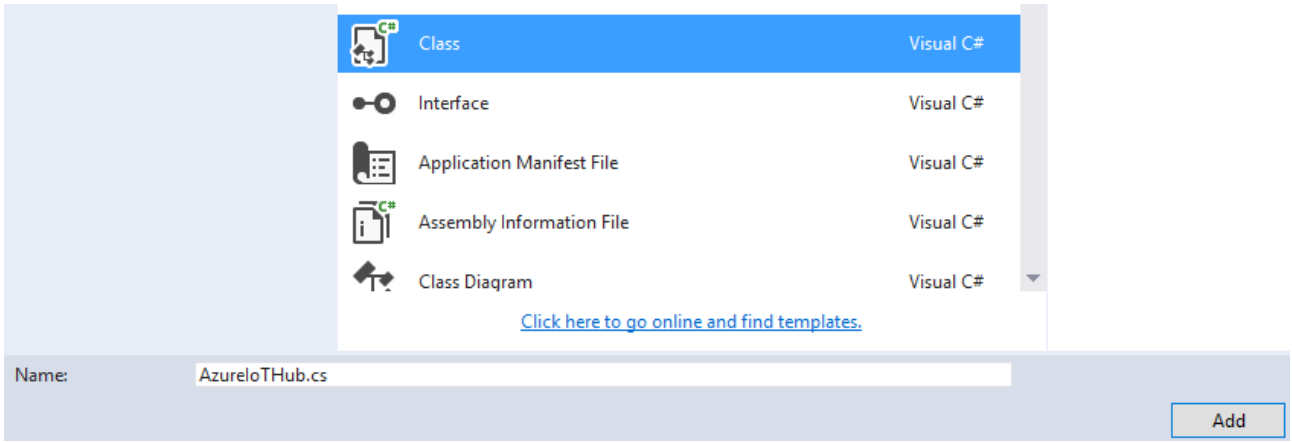


Fig.342

5.3.3.1 TX20.cs

Questa classe, una volta istanziata nello HEAP come oggetto, ha il ruolo di contenitore dei dati, più precisamente deve memorizzate la velocità e direzione vento, la data e l'ora della singola lettura. L'oggetto con i dati verrà poi serializzato in formato JSON ed inviato al cloud. L'oggetto verrà quindi deallocato dalla memoria da parte del garbage collector. Il ciclo riparte con una seconda, terza ed n-esima lettura. Segue il codice della classe.

```
namespace StazioneMeteo
{
    class TX20
    {
        public float VelocitaMS { get; set; }
        public string Direzione { get; set; }
        public string Data { get; set; }
        public string Ora { get; set; }

        public TX20()
        {
            VelocitaMS = 0;
            Direzione = "-";
            Data = "-";
            Ora = "-";
        }

        public override string ToString()
        {
            return $"AnemometroTX20:: Velocità: {VelocitaMS}, Direzione: {Direzione}, Data:
                    {Data}, Ora: {Ora}";
        }

    }
}
```

I metodi "getter" e "setter" permettono l'implementazione delle proprietà di classe, quindi non serve dichiarare in modo esplicito i dati membro, come avviene in Java. Il costruttore di classe di default effettua l'inizializzazione dei campi. Il metodo "ToString()" viene ridefinito, permettendo di stampare un stringa con tutti i valori rilevati dal Raspberry, comodo in fase di Debug per verificare la correttezza dei dati rilevati, che ovviamente devono essere identici a quelli visualizzati sul display, se presente. Il simbolo "$" davanti alla stringa serve per considerare i caratteri secondo lo standard Unicode, quindi è possibile inserire caratteri non compresi nella codifica ASCII.

5.3.3.2 TX20Exception.cs

Il codice C# di questa classe realizza l'eccezione personalizzata "TX20Exception".

```
using System;

namespace StazioneMeteo
{
   class TX20Exception : Exception
   {
      private string messaggioPersonalizzato;

       public TX20Exception(string messaggio)
       {
          messaggioPersonalizzato = messaggio;
       }

       public override string Message
       {
          get   {   return $"TX20! " + messaggioPersonalizzato + " " + base.Message;   }
       }
   }
}
```

Quando si vuole creare una propria eccezione, è necessario estendere la classe statica "Exception", ridefinire il metodo "Message" ed impostare il costruttore di classe.

5.3.3.3 AzureIoTHub.cs

Questa classe è fondamentale per la connettività ad Azure, definendo il metodo "inizializza" che provvede ad instaurare una connessione verso il servizio "IoT Hub", tramite il nome del servizio stesso, tramite il nome del device registrato nel Hub e tramite la chiave di connessione generata automaticamente in "IoT Hub". Questo metodo sfrutta la "Create" della classe statica "DeviceClient", richiedendo come parametro formale il nome del servizio, un oggetto di tipo "DeviceAuthenticationWithRegistrySymmetricKey" che a sua volta richiede nome del device registrato e chiave, infine serve indicare, tramite il tipo enumerativo "TransportType", il protocollo che si vuole utilizzare. Qualora la connessione vada a buon fine, il contenuto del dato membro private "device" non sarà nullo, quindi viene inizializzato il dato membro " inizializzazioneOK" a "true", altrimenti " false". Il dato membro verrà utilizzato per verificare se vi è connettività verso il cloud, non per nulla è presente il metodo "isInizializzato". La verifica di connettività avviene all'interno del blocco "try-catch" che intercetta l'eccezione "Exception" richiamando, se sollevata, l'eccezione personalizzata "TX20Exception". Il metodo "inviaDatiAlCloud" realizza in primis la serializzazione dei dati presenti nell'oggetto di tipo "TX20" nel formato JSON, sfruttando il metodo "SerializeObject" della classe statica "JsonConvert". L'output serializzato è una stringa, e viene memorizzato nel dato membro "datiJSON". Lo step successivo è l'invio della stringa JSON, ossia "datiJSON", al cloud. Per fare questo si deve creare un oggetto di tipo "Message", tramite la codifica UTF8 della stringa serializzata "datiJSON". Il messaggio viene salvato nel dato membro "datiJSONVersoAzure" ed inviato al cloud tramite il metodo "SendEventAsync" del dato membro "pi3" di tipo "DeviceClient" precedentemente inizializzato. L'invio avviene tramite la clausola "await" perché si vuole che la chiamata non sia bloccante, ma che venga eseguita in modo asincrono. Nel momento in cui si utilizza questa clausola, l'intero metodo "inviaDatiAlCloud" diviene non bloccante, quindi nella firma serve inserire la parola chiave "async". E' interessante notare che "inviaDatiAlCloud" restituisce un dato di tipo "Task", questo perché potrebbe essere interessante verificare se il task di invio dei dati al cloud è stato o meno completato. Le operazioni di serializzazione ed invio dei dati vanno effettuate all'interno del blocco "try-

catch", verificando l'eccezione "Exception" e sollevando, eventualmente, quella personalizzata "TX20Exception".

```
using System;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;

namespace StazioneMeteo
{
   class AzureIoTHub
   {
      private DeviceClient pi3;
      private Microsoft.Azure.Devices.Client.Message datiJSONVersoAzure;
      private string datiJSON;
      private bool inizializzazioneOK;

      public bool isInizializzato
      {
        get    {   return inizializzazioneOK;   }
       }

      public void inizializza(string IotHubHostName, string deviceKey, string deviceId)
      {
         inizializzazioneOK = false;

        try
        {
           pi3 = DeviceClient.Create(IotHubHostName,
               new DeviceAuthenticationWithRegistrySymmetricKey(deviceId, deviceKey),
```

```
            Microsoft.Azure.Devices.Client.TransportType.Http1);

            if (pi3 != null)
            {
                inizializzazioneOK = true;
            }
        }
        catch (Exception ex)
        {
            Debug.WriteLine("Connessione IoT Hub fallita! " + ex.Message);

            throw new TX20Exception(ex.Message);
        }
    }

    public async Task inviaDatiAlCloud(TX20 DatiDelTX20)
    {
        try
        {
            datiJSON = JsonConvert.SerializeObject(DatiDelTX20)
            datiJSONVersoAzure = new  Microsoft.Azure.Devices.Client.Message
                                        (
                                            Encoding.UTF8.GetBytes(datiJSON)
                                        );
            await pi3.SendEventAsync(datiJSONVersoAzure);
        }
        catch (Exception ex)
        {
            Debug.WriteLine("Invio dei dati fallito! " + ex.Message);
            throw new TX20Exception(ex.Message);
        }
    }
}
```

5.3.3.4 MainPage.xaml.cs

Il codice di questa classe rimane identico a quello discusso nel capitolo precedente, con l'ovvia aggiunta del codice per inviare i dati al cloud tramite le classi viste nei paragrafi precedenti. Viene riportato il codice semi completo, con la sola definizione dei metodi già visti e discussi privi di codice e le parti nuove per completare definitivamente la UWP. I nuovi dati membro sono "iotHubHostname", "deviceName", "deviceKey", "IoTHub" e "anemometroTX20". Il dato "iotHubHostname" definisce il nome del servizio Azure "IoT Hubs" utilizzato in fase di configurazione, che corrisponde a "RaspberryTX20-IoTHub.azure-devices.net", quindi un nome FQDN completo che verrà risolto in indirizzo IP a livello di DNS. Il dato "deviceName" corrisponde al nome del device Raspberry registrato nel servizio "IoT Hubs", e corrisponde a "Raspberry1_TX20". Il dato "deviceKey" è la chiave primaria generata automaticamente e necessaria per la connessione al servizio. Il dato membro "IoTHub", di tipo classe "AzureIoTHub", serve per richiamare i metodi di inizializzazione della connessione e di trasferimento, come discusso nel paragrafo precedente. Per concludere, il dato membro "anemometroTX20" è un oggetto di tipo classe "TX20" che contiene le informazioni sulla singola lettura. Oltre a questi nuovi dati membro, vengono utilizzati due nuovi metodi privati, chiamati "setupIoTHub()" e "inviaDatiAlCloud()". Il primo metodo, come si evince dal codice sottostante, non fa altro che istanziare l'oggetto "IoTHub" per poi effettuare l'inizializzazione della connettività al servizio di Azure tramite il metodo "inizializza" al quale passa i dati membro "iotHubHostname", "deviceName" e "deviceKey". Il metodo "trasferisciDatiAlCloud()" non fa altro che richiamare il metodo "inviaDatiAlCloud()" passando l'oggetto di tipo classe "TX20" che contiene i dati della lettura, così che possa subire il meccanismo di serializzazione nel formato JSON, con conseguente creazione del messaggio per l'invio, non bloccante, al cloud. Essendo la "inviaDatiAlCloud()" di tipo asincrono, serve anteporre la clausola "await" davanti al nome del metodo. Il costruttore di classe "MainPage()" viene eseguito all'avvio della UWP, quindi è il posto più comodo per creare l'oggetto di connettività al cloud "IoTHub", verificando che la connessione sia avvenuta con successo. Qualora per motivi sconosciuti non sia possibile connettersi al cloud Azure, la UWP non effettua nessuna lettura dall'anemometro TX20. Questa scelta è abbastanza drastica, ma nulla vieta di implementare una soluzione più soft che, tramite un blocco "try-catch", verifichi in fase di acquisizione dei dati dal TX20, la disponibilità o meno del servizio ed, in caso positivo, provvedere all'invio dei dati. Nel momento in cui è presente la connettività

ad Azure, verificabile con il metodo "isInizializzato", il costruttore "MainPage()" crea l'oggetto "anemometroTX20" e richiama il metodo "attivazioneTX20()" come già discusso nel capitolo 4. L'ultimo passo consiste nel modificare lo stato dell'oggetto "anemometroTX20", tramite la memorizzazione dei dati della singola lettura, operazione che avviene all'interno del metodo "letturaDatiDaTX20()", sfruttando le proprietà della classe "TX20" e richiamando il metodo privato "trasferisciDatiAlCloud()". Per agevolare il lettore nell'individuare le nuove parti di codice, vengono inserite le nuove righe all'interno di un riquadro.

```
#define NOPRINT
using System;
using Windows.UI.Xaml.Controls;
using Windows.Devices.Gpio;
using Windows.UI.Core;
using System.Diagnostics;

namespace StazioneMeteo
{
   public sealed partial class MainPage : Page
   {
      private const int ritardoBitMicroSecondi = 1220;
      private const int ritardoDatagrammaMicroSecondi = 1000000;
      private const int pinRicezioneTX20 = 4;
      private const int pinAttivazioneTrasmissioneTX20 = 17;
      private GpioController gpio;
      private GpioPin pinGPIO4;
      private GpioPin pinGPIO17;

      String[] elencoDirezioni = new String[16] {
                                    "N","NNE","NE","ENE","E","ESE","SE","SSE",
                                    "S","SSW","SW","WSW","W","WNW","NW","NNW"
                                   };
```

```
private string iotHubHostname = "RaspberryTX20-IoTHub.azure-devices.net";
private string deviceName = "Raspberry1_TX20";
private string deviceKey = "MIA CHIAVE!!";
private AzureIoTHub IotHub;
private TX20 anemometroTX20;
```

```
public MainPage()
{
    this.InitializeComponent();
```

```
    setupIoTHub();

    if (this.iotHub.isInizializzato)
    {
        this.anemometroTX20 = new TX20();
        attivazioneTX20();
    }
}
```

```
public async void trasferisciDatiAlCloud()
{
    await this.IotHub.inviaDatiAlCloud(anemometroTX20);
}

private void setupIoTHub()
{
    this.IotHub = new AzureIoTHub();
    this.IotHub.inizializza(iotHubHostname, deviceKey, deviceName);
}
```

```
private void attivazioneTX20()  { ... }
```

```
private void letturaDatiDaTX20(GpioPin sender, GpioPinValueChangedEventArgs e)
{
  var task = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
  {
    if (e.Edge == GpioPinEdge.FallingEdge)
    {
       pinGPIO4.ValueChanged -= letturaDatiDaTX20;

       int bitsStartFrame = startFrame();
       int bitsDirezioneVento1 = direzioneVento1();
       int bitsVelocitaVento1 = velocitaVento1();
       int bitsChecksum = checksum();
       int bitsDirezioneVento2 = direzioneVento2();
       int bitsVelocitaVento2 = velocitaVento2();

       int calcoloChecksum = calcoloDelChecksumSulDatagramma
                                    (bitsDirezioneVento1, bitsVelocitaVento1);

        if (
              bitsStartFrame == 4 && bitsDirezioneVento1 == bitsDirezioneVento2 &&
              bitsVelocitaVento1 == bitsVelocitaVento2 &&
              calcoloChecksum == bitsChecksum
           )
       {
          string direzioneDelVento = elencoDirezioni[bitsDirezioneVento1].ToString();

          Debug.Write("------------------------------------------------------------\n");
          Debug.Write("La direzione del vento è = " + direzioneDelVento);
          Debug.WriteLine(" - La velocità del vento è pari a {0} m/s ({1} km/h - {2} mph)",
              bitsVelocitaVento1.ToString(), ((float)bitsVelocitaVento1 * 3.6).ToString(),
              ((float)bitsVelocitaVento1 * 2.2).ToString());
          Debug.WriteLine("------------------------------------------------------------");
          Debug.WriteLine("////////////////////////////////////////////////////////////");
```

```
            txtDirezioneVento.Text = "Direzione vento = " + direzioneDelVento;
            txtVelocitaMS.Text = bitsVelocitaVento1.ToString() + " m/s";
            txtVelocitaKMH.Text = ((float)bitsVelocitaVento1 * 3.6).ToString() + " km/h";
            txtVelocitaMPH.Text= ((float)bitsVelocitaVento1 * 2.2).ToString() + " mph";
            this.anemometroTX20.Direzione = direzioneDelVento;
            this.anemometroTX20.VelocitaMS = (float)bitsVelocitaVento1;
            this.anemometroTX20.Data = DateTime.Today.ToString("dd-MM-yyyy");
            this.anemometroTX20.Ora = DateTime.Now.ToString("HH-mm-ss");

            trasferisciDatiAlCloud();
          }
          delay(ritardoDatagrammaMicroSecondi);
          pinGPIO4.ValueChanged += letturaDatiDaTX20;
        }
      }
    );
  }

  private bool inizializzazioneGPIO()  { ... }
  int startFrame()  { ... }
  int direzioneVento1()  { ... }
  int direzioneVento2()  { ... }
  int velocitaVento1()  { ... }
  int velocitaVento2()  { ... }
  int checksum()  { ... }
  int calcoloDelChecksumSulDatagramma
                                        (
                                          int bitsDirezioneVento1,
                                          int bitsVelocitaVento1
                                        ) { ... }
  void delay(int microSecondiDiRitardo)  { ... }
 }
}
```

5.4 *Lettura dati dal cloud con web app MVC 5*

Una volta che i dati raggiungono il servizio "IoT Hubs" chiamato "RaspberryTX20-IoTHub.azure-devices.net", finiscono, grazie al modulo "Stream Analytics job", nello storage BLOB chiamato "dati". E' necessario dare la possibilità di visualizzare queste informazioni garantendo l'indipendenza dalla piattaforma. La scelta più semplice ricade sull'utilizzo di un browser internet, motivo per cui Azure offre la possibilità di realizzare delle Web App come estensione del classico web site. Quando un cliente necessità di realizzare delle app per smartphone, ha il problema di supportare le tre tecnologie dominanti, iOS di Apple, Android e Windows Phone. Detto in altre parole, tre realizzazioni diverse del medesimo progetto. Con una Web App è possibile scrivere il codice una ed una sola volta facendolo girare sul cloud Azure, in questo modo l'unico strumento necessario per la navigazione è proprio il browser internet, sia esso Safari, Chrome, Firefox o Explorer.

5.4.1 MVC

Oggi giorno è molto facile realizzare siti web per le più svariati attività, un semplice sito di e-commerce è facilmente personalizzabile anche da parte di personale non qualificato, tramite l'impiego di framework dedicati a questo scopo. Gli aspetti di progettazione non interessano l'utente finale, ma sono il punto di partenza per qualsiasi specialista di settore, visto che i framework utilizzati nel web sfruttano metodologie di gestione delle pagine ben precise. Il cardine fondamentale di questo discorso sono proprio le pagine web e quindi le richieste distribuite che arrivano dai client della rete internet al sito web. Ogni richiesta è ovviamente asincrona, quindi la gestione delle interfacce web diviene complessa e rischia di esplodere nel tentativo di tenere traccia di ciò che succede, al punto di rendere l'intero sistema ingestibile. In simili situazioni è necessario implementare una progettazione che si basi su una macchina a stati finiti, capace, indipendentemente dal numero delle richieste distribuite asincrone alle interfacce web, di permettere l'avanzamento corretto della navigazione, senza chiamare in causa interfacce web non corrette. Questo approccio di progettazione trova risposta nel paradigma MVC "Model-View-Controller" e mira a stratificare l'applicazione web così che il passaggio da uno stato all'altro della macchina a stati finiti sia efficiente ed immediato. Una logica stratificata permette, allo strato X, di usufruire delle funzioni offerte dallo strato X-1, ossia quello sottostante, quest'ultimo infatti

ha il compito di passare i dati allo strato X, cioè quello gli sta subito sopra. Questa logica di "offerta/passaggio dati" tra due strati adiacenti, permette agli sviluppatori di separare le responsabilità, concentrandosi esclusivamente sullo strato di competenza. Per un'applicazione web la separazione delle responsabilità, e quindi del lavoro, è un fattore chiave per la realizzazione di progetti molto complessi. Un altro grande vantaggio della logica a strati, è la possibilità di riutilizzare il medesimo strato in parti diverse dell'applicazione. Il paradigma MVC offre la possibilità di sviluppare a strati. Una prima rappresentazione della logica MVC è quella di Fig.343.

Fig.343

Il browser del client effettua una richiesta distribuita asincrona HTTP_REQUEST al sito web, rappresentato dal blocco "WEB TIER", il quale ha in loco la pagina dinamica priva di dati, ne consegue quindi una REQUEST verso il blocco chiamato "DATA TIER" che estrae le informazioni, per esempio da un database, inviando la RESPONSE esclusivamente al "WEB TIER". Quest'ultimo prepara la pagina web richiesta con i dati e la invia come HTTP_RESPONSE al browser che ne ha fatto richiesta. In questo schema non sono evidenziate le componenti del paradigma MVC, quindi una revisione più dettagliata dell'interazione con l'applicazione web è raffigurata nello schema di Fig.344.

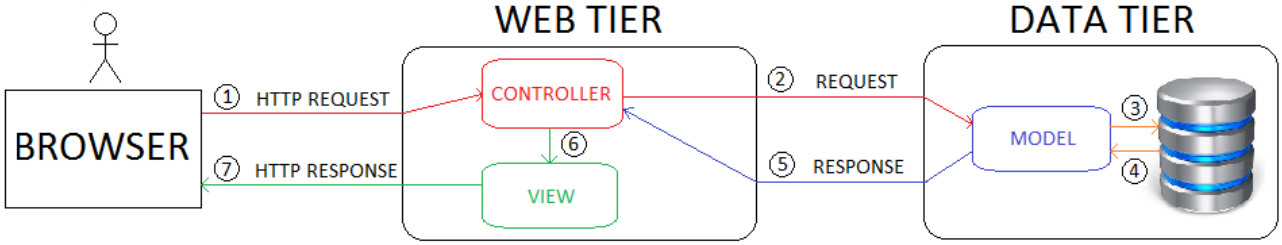


Fig.344

Come sempre l'attore principale è il browser, che effettua una richiesta HTTP_REQUEST indicando l'azione che vuole compiere, indicando il metodo HTTP e la pagina desiderata.

Tutte le richieste HTTP arrivano sempre al blocco "Controller" che funge da strato di raccordo, avente il compito di decidere quale vista mandare al client, dopo aver opportunamente richiesto ed ottenuto i dati da inserire nella vista stessa. La vista quindi è la pagina web dinamica con i dati. E' compito del "Controller" decidere quale vista selezionare e quale tipo di richiesta dati inviare al blocco "DATA TIER". In questo blocco le richieste che provengono dal "Controller", arrivano al "Model" che ha il compito di prelevare i dati richiesti (totalmente o parzialmente) da un database, sia esso relazione o NoSQL, inviandoli conseguentemente al "Controller" che così potrà decidere quale vista utilizzare popolandola con i dati appena ricevuti. La vista "View" è quindi la pagina web o i dati in formato JSON/XML che verranno mandati al client tramite una HTTP_RESPONSE. Questa è l'interazione classica che avviene in un'applicazione web che sfrutta MVC. Da un punto di vista del linguaggio di programmazione, il "Controller" è un oggetto che ha il compito di interagire con l'utente e di mettere in comunicazione l'oggetto "View" con l'oggetto "Model", in altri termini è il fulcro dell'interazione "Modello-Vista". Un oggetto "Model" viene realizzato per accedere al database, indipendentemente dalla tecnologia di quest'ultimo, per poi veicolare i dati estrapolati al "Controller". La "View" rappresenta un pool di oggetti, ed è compito del "Controller" decidere la vista corretta inviare al client popolandola con le informazione ricevute dal "Model". A parità di modello possono esistere più viste. L'ordine di progettazione di questi tre blocchi è fondamentale, per cui conviene procedere con la seguente sequenza:

1. Model
2. View
3. Controller

Lo sviluppatore deve scrivere per prima la classe che rappresenta il modello, solo così avrà la certezza di avere sotto controllo tutte le chiamate al database, visto che i dati sono il cuore e l'anima di qualsiasi applicazione di business. Le classi successive sono tutte quelle che rappresentano le viste, il più delle volte pagine web, ma anche dati in formato JSON o XML. Esistono molti linguaggi di programmazione che permettono lo sviluppo di pagine web, i più diffusi al mondo sono PHP, JSP e ASP.NET. Un oggetto vista serve per permettere l'interazione visuale delle informazioni presenti nel blocco "DATA TIER". Una vista può solo visualizzare i dati, ma non modificare le informazione del database.

L'interazione tra oggetti vista ed oggetto modello, non è mai diretto, ma viene normato dall'oggetto controllore. Un'applicazione web potrebbe avere anche un sola classe modello, ma più classi vista che visualizzeranno dati diversi del database. L'ultima classe da realizzare è il controllore, con lo scopo di decidere quali dati richiedere al modello e quali pagine utilizzare per la visualizzazione.

5.4.2 WebApp con Visual Studio 2015

Conoscere il paradigma MVC è un requisito cardine prima di iniziare a sviluppare qualsiasi applicazione web, indipendentemente dal linguaggio e dall'ambiente di sviluppo a disposizione. La scelta di C# ASP.NET è legata all'ambiente Visual Studio di Microsoft e allo stesso cloud Azure. Prima di strutturare l'applicazione web secondo il paradigma MVC, è necessario creare un progetto web app in modo corretto, per poi iniziare la scrittura della classe modello, vista e controllore. Aprire una sessione di Visual Studio e creare un nuovo progetto come da Fig.345.

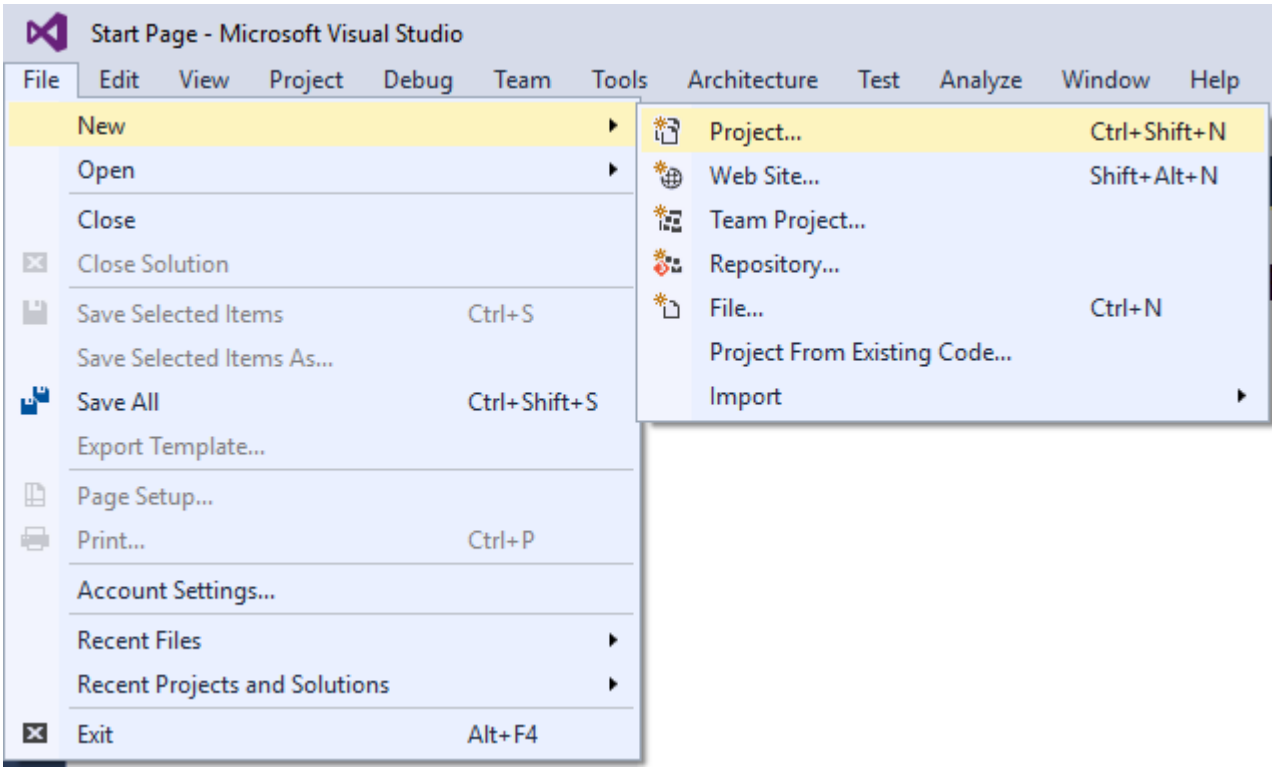


Fig.345

Selezionare "Visual C# ► Cloud ► ASP.NET Web Application (.NET Framework)". Specificare la directory ed il nome del progetto desiderato, in questo caso "WebAppTX20", come da Fig.346. Premere "OK" per confermare. In rosso l'account Microsoft utilizzato per registrare il codice di attivazione di Visual Studio.

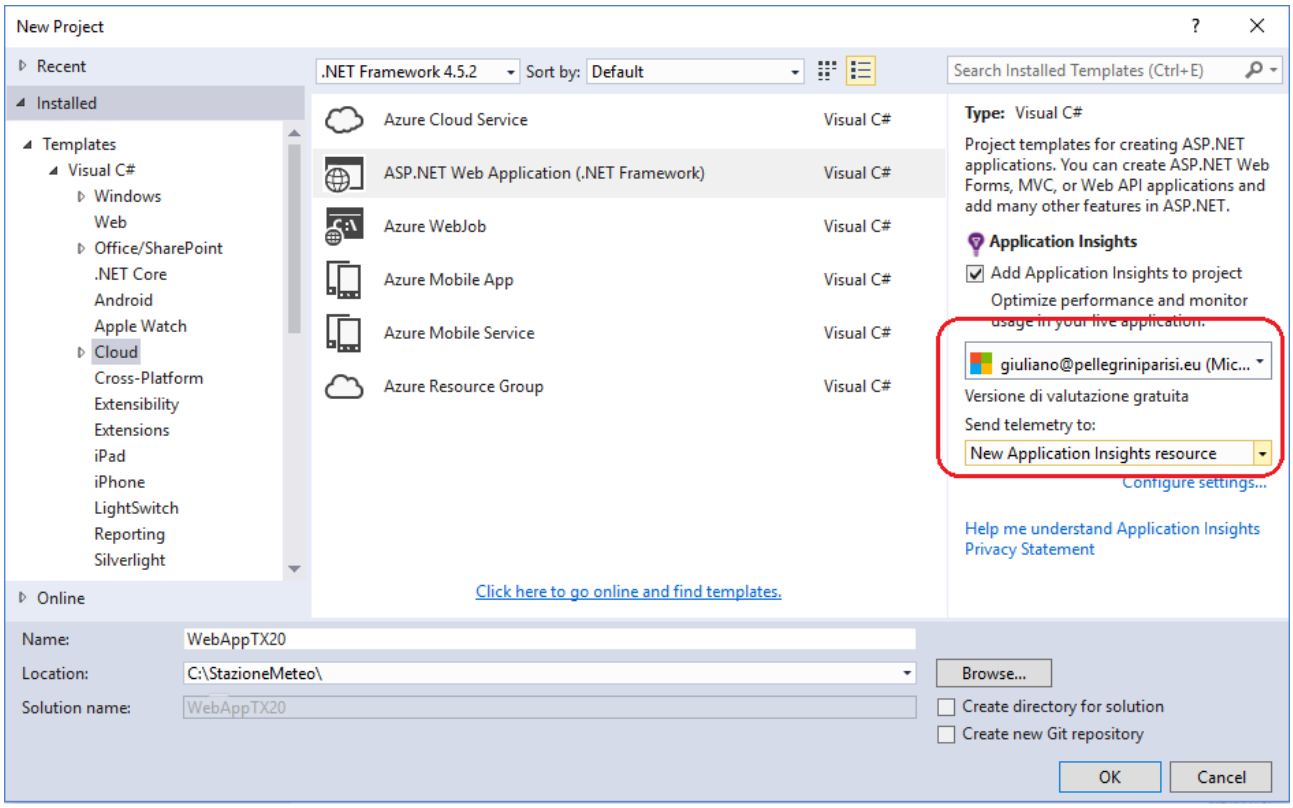


Fig.346

La schermata successiva presenta la scelta del modello di progettazione web che si desidera utilizzare. La scelta ricade su MVC, ma si potrebbe anche impiegare un approccio di tipo "Web Forms" in presenza di sviluppatori non abituati a stratificare il progetto con conseguente separazione delle responsabilità. Questo aspetto è l'elemento portante di tutto il paradigma MVC, che in "Web Forms" non trova riscontro. Un altro aspetto molto interessante è che con MVC lo sviluppatore deve realizzare da se tutta la vista con codice HTML, mentre in "Web Forms" non è necessario. Quest'aspetto è quindi molto importante, sopratutto quando si vuole realizzare qualcosa di personalizzato che impiega codice server-side come Javascript. In questo caso la scelta di MVC è obbligata, come infatti avviene in questo progetto per la scelta di impiegare Javascript per graficare i dati sul web. Un altra scelta interessante è "Web API", che nel momento in cui si impiega

".NET Core 1.0", è di per se la medesima cosa di MVC, visto che entrambi i modelli permettono l'impiego di un solo controllore dello stesso tipo. Utilizzare una "Web API" e di conseguente un "API Controller", permette di organizzare meglio il progetto e di essere, molto probabilmente, più idoneo per restituire dati al client sotto forma di JSON o XML. Nel momento in cui invece si vuole restituire pagine web, conviene appoggiarsi a MVC. In ogni caso la differenza tra i due modelli è molto sottile con ".NET Core". Il modello "Azure Mobile App" (AWS) non è molto diverso da "Web API" ed è adatto quando si vuole utilizzare il servizio di Azure chiamato "Active Directory". Questi in sintesi i modelli. In Fig.347 si vede l'impostazione di MVC e una richiesta di autenticazione basata su account, oltre che l'impostazione del flag "Host in the cloud" e la scelta un servizio "App service", così da rendere pubblica la web application. Premere "Change Autentication" e modificare il tipo di autenticazione in "No Authentication" come in Fig.348.

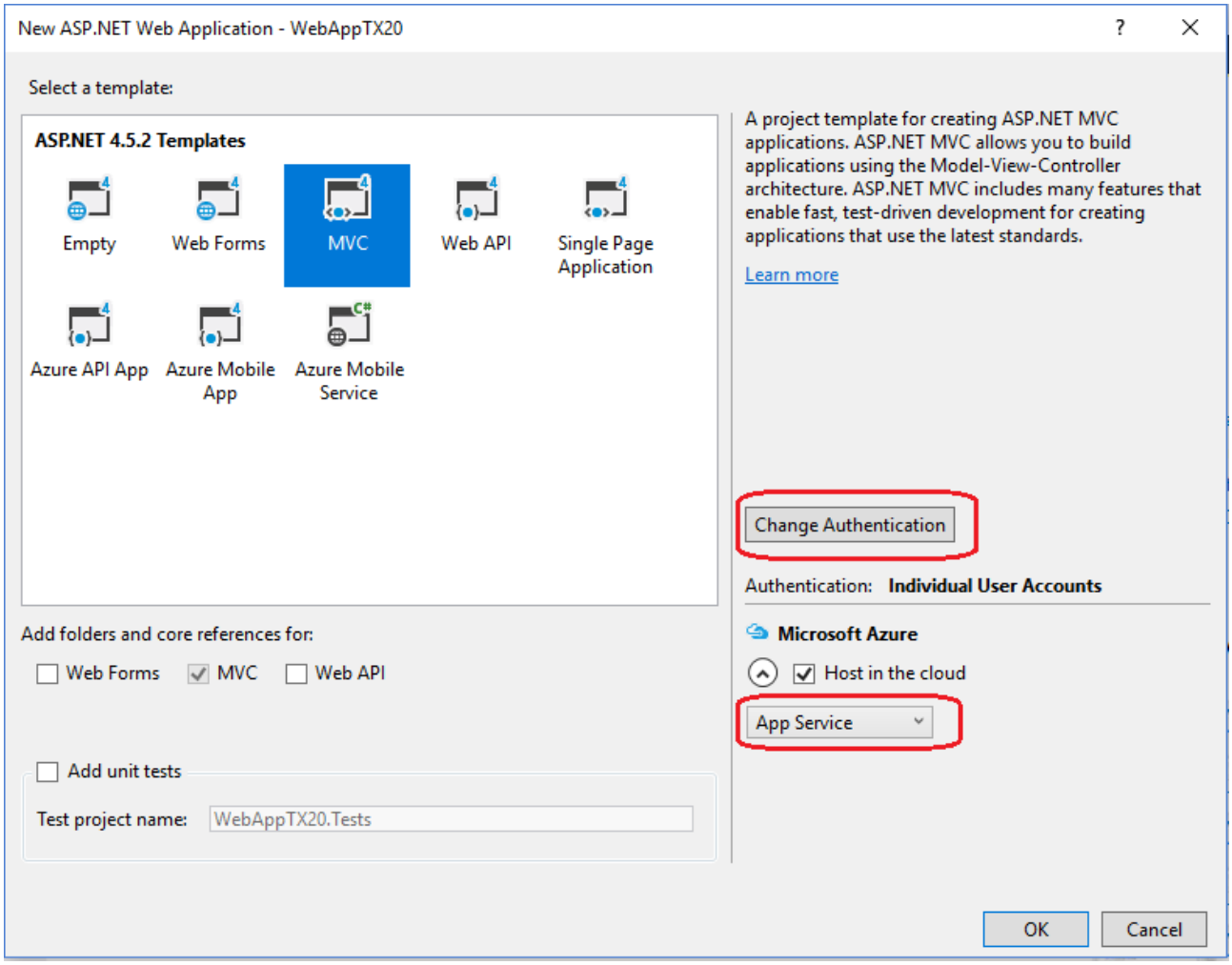


Fig.347

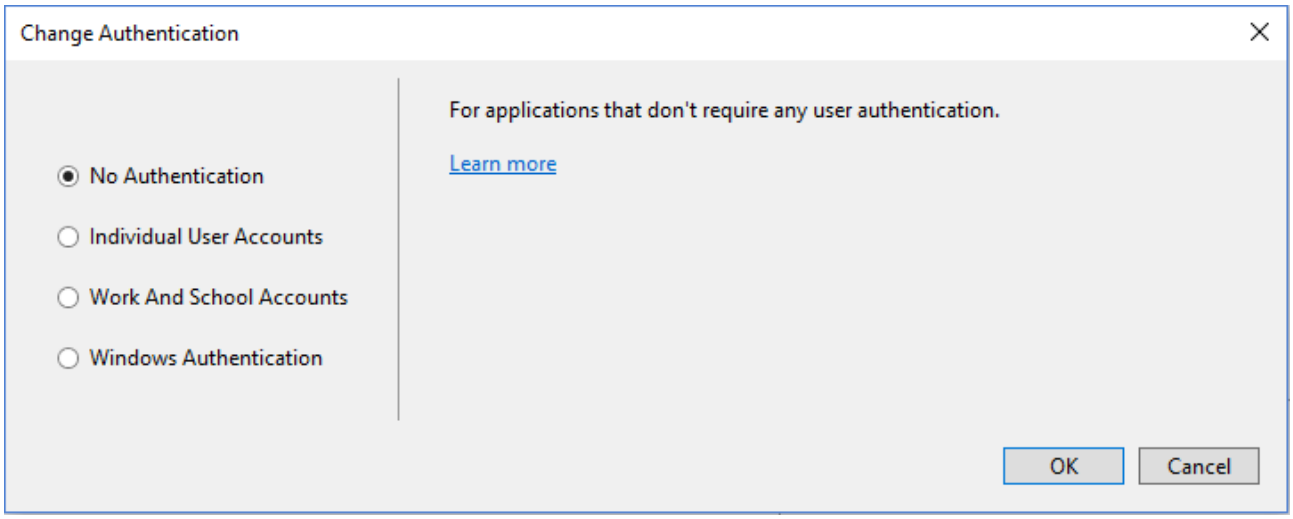


Fig.348

Confermare la web application premendo OK in modo da accedere alla configurazione finale della "App Service", come da Fig.349.

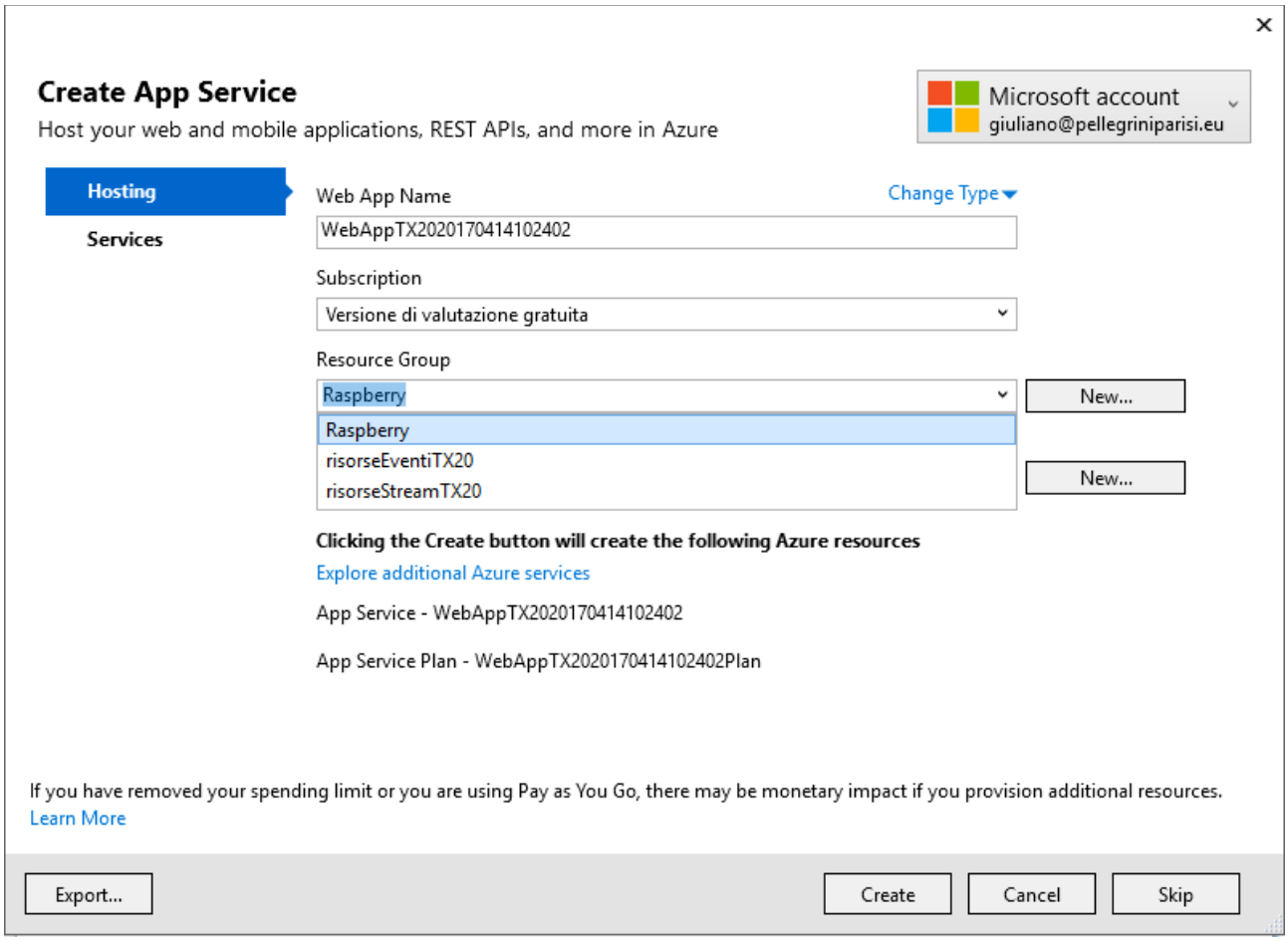


Fig.349

In automatico viene proposta la "Web App Name", così come il tipo di sottoscrizione gratuita. Selezionare nella listbox "Resource Group" la voce "Raspberry" che corrisponde al nome delle risorse create dal servizio "RaspberryTX20-IoTHub" configurato in precedenza. Il passo finale è confermare la web app premendo il pulsante "Create", che attiva il meccanismo di creazione del progetto "WebAppTX20" come da Fig.350.

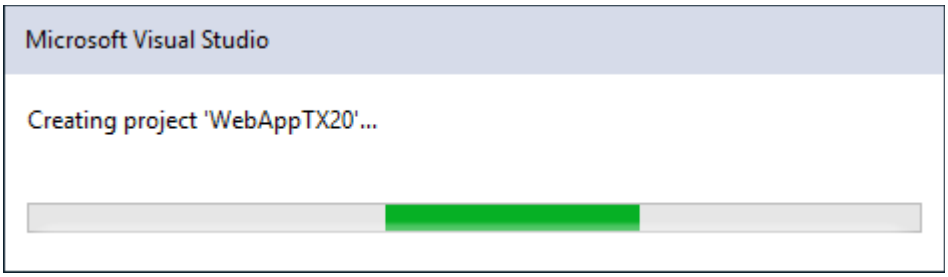


Fig.350

Terminata la creazione del progetto ne viene data conferma come da Fig.351.

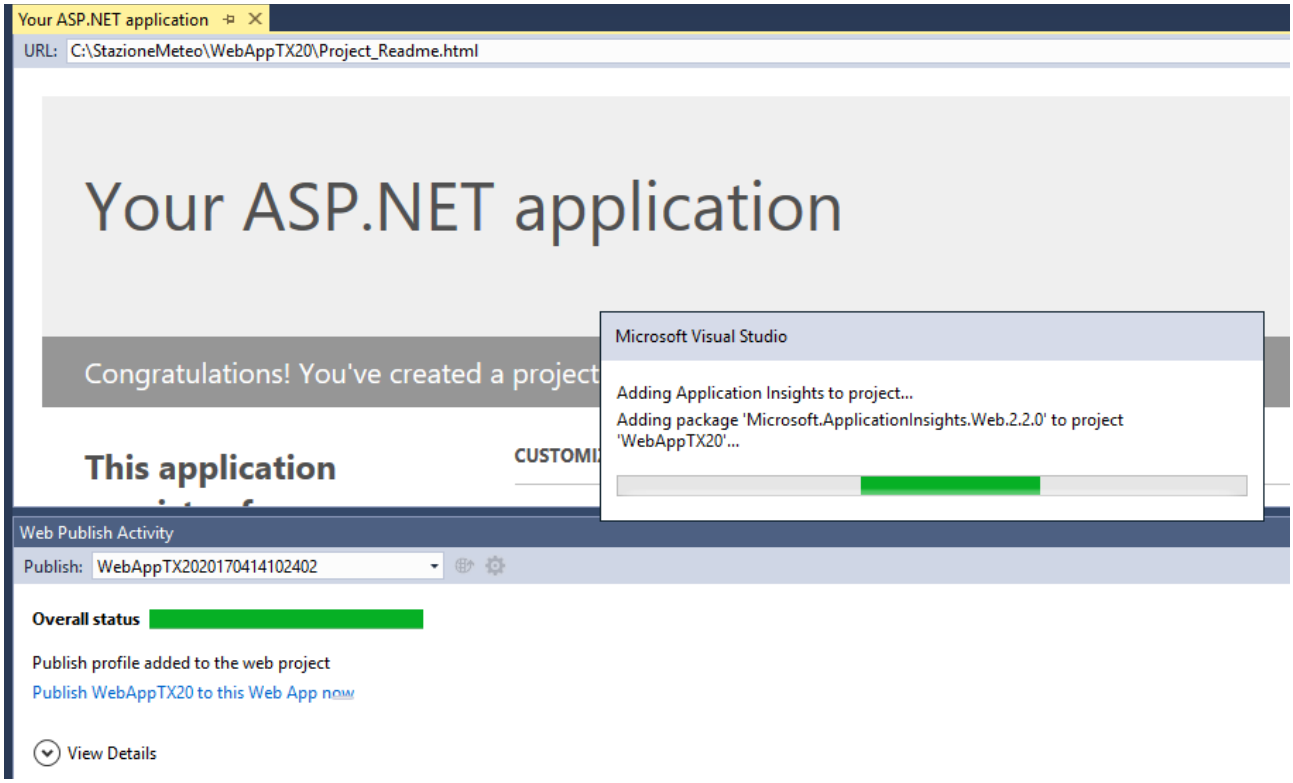


Fig.351

La struttura del progetto C# ASP.NET MVC è definita nella "Solution Explorer" di Fig.352.

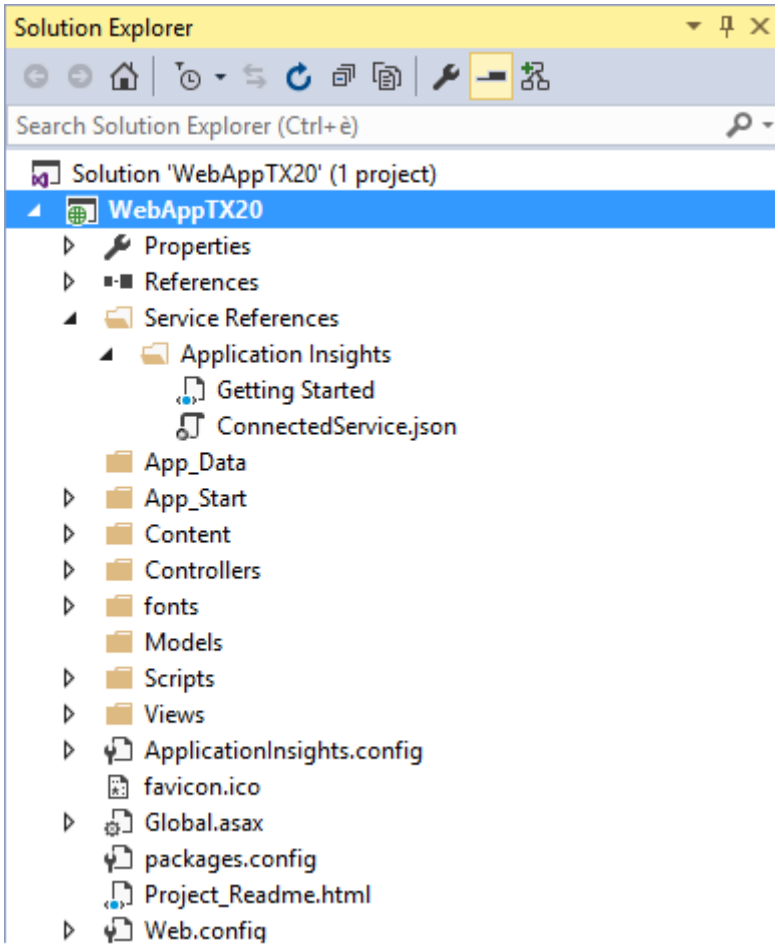


Fig.352

E' interessante notare la presenza delle seguenti directory/file:

- Models dove si creano le classi per la realizzazione del modello
- Views dove si creano le pagine web HTML
- Controllers dove si creano le classi per la realizzazione dei controller
- Scripts dove si creano/caricano gli script javascript
- Content dove si caricano le immagini e i fogli di stile CSS
- Web.config file di configurazione della web app

Prima di iniziare a scrivere una sola istruzione C#, conviene configurare l'applicazione web per la connessione al servizio di Azure "Raspberry-TX20, inserendo un apposito tag nel file di configurazione "Web.config".

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="connAzure"
connectionString="DefaultEndpointsProtocol=https;AccountName=tx20;AccountKey=0f/4b
**********************************************************************************************************
***************5boCXh/bQ==;EndpointSuffix=core.windows.net" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
   ecc....
```

Selezionando il nome del progetto "WebAppTX20" e cliccando il tasto destro del mouse, è possibile procedere alla pubblicazione dell'applicazione web sul cloud tramite la voce "Publish..." di Fig.353.

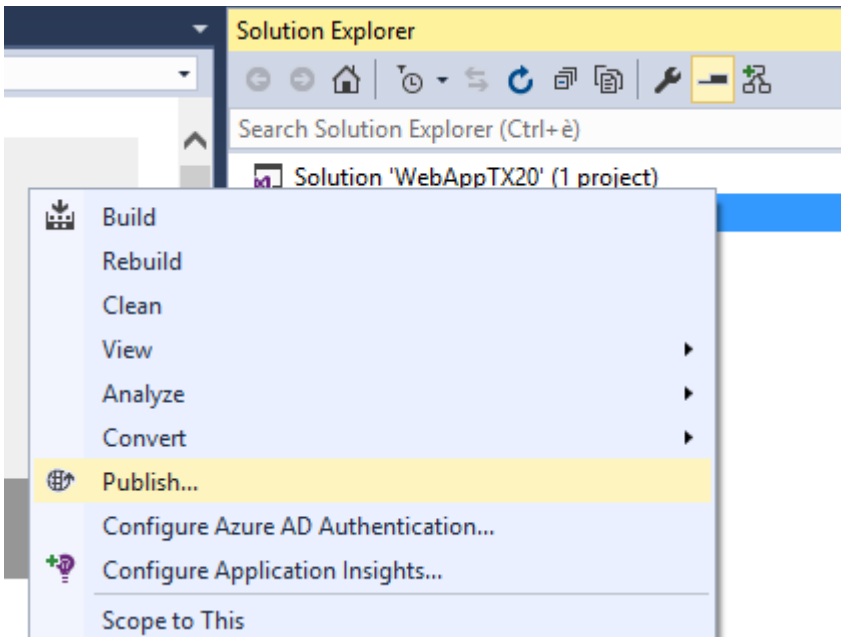


Fig.353

Viene riportato il nome pubblico dell'applicazione che dovrà venire specificato nella URL da inserire nel browser. In questo caso "WebAppTX2020170414102402" come da Fig.354. E' possibile effettuare un preview di tutti i files che verranno pubblicati nel cloud, tramite l'apposito pulsante "Start Preview", come da Fig.355.

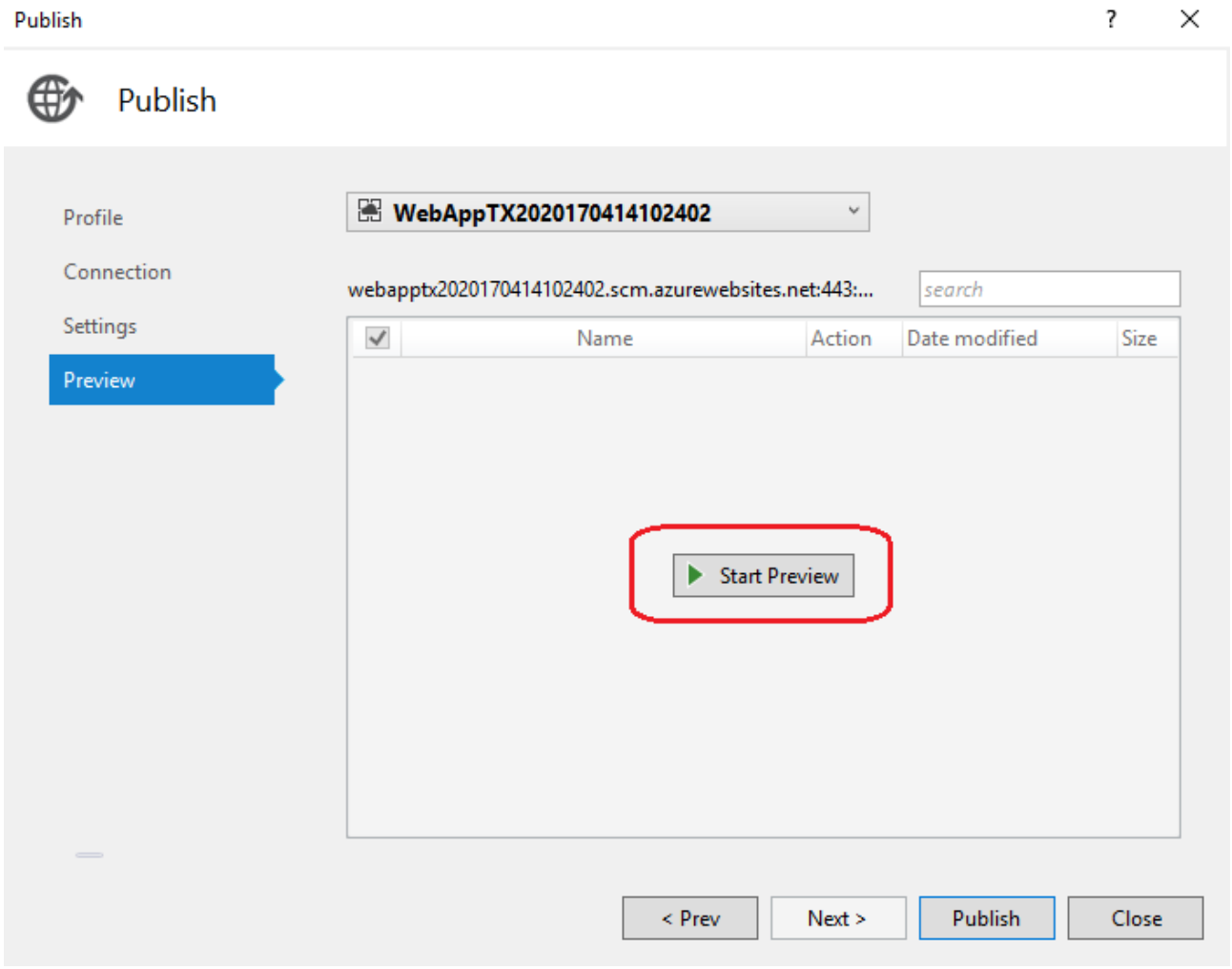


Fig.354

La fase di preview non è obbligatoria, ma è interessante notare la colonna "Action" nella quale si vede chiaramente l'azione di tipo "Add" di eventuali nuovi files. Successive pubblicazioni inseriscono, aggiornano o rimuovono solo i nuovi files o quelli modificati. Premere il pulsante "Publish" per avviare l'operazione di pubblicazione. Terminato il trasferimento dei files, nella sezione di "Output" viene riportato il link completo per accedere remotamente all'applicazione web, come si evince dalla Fig.356.

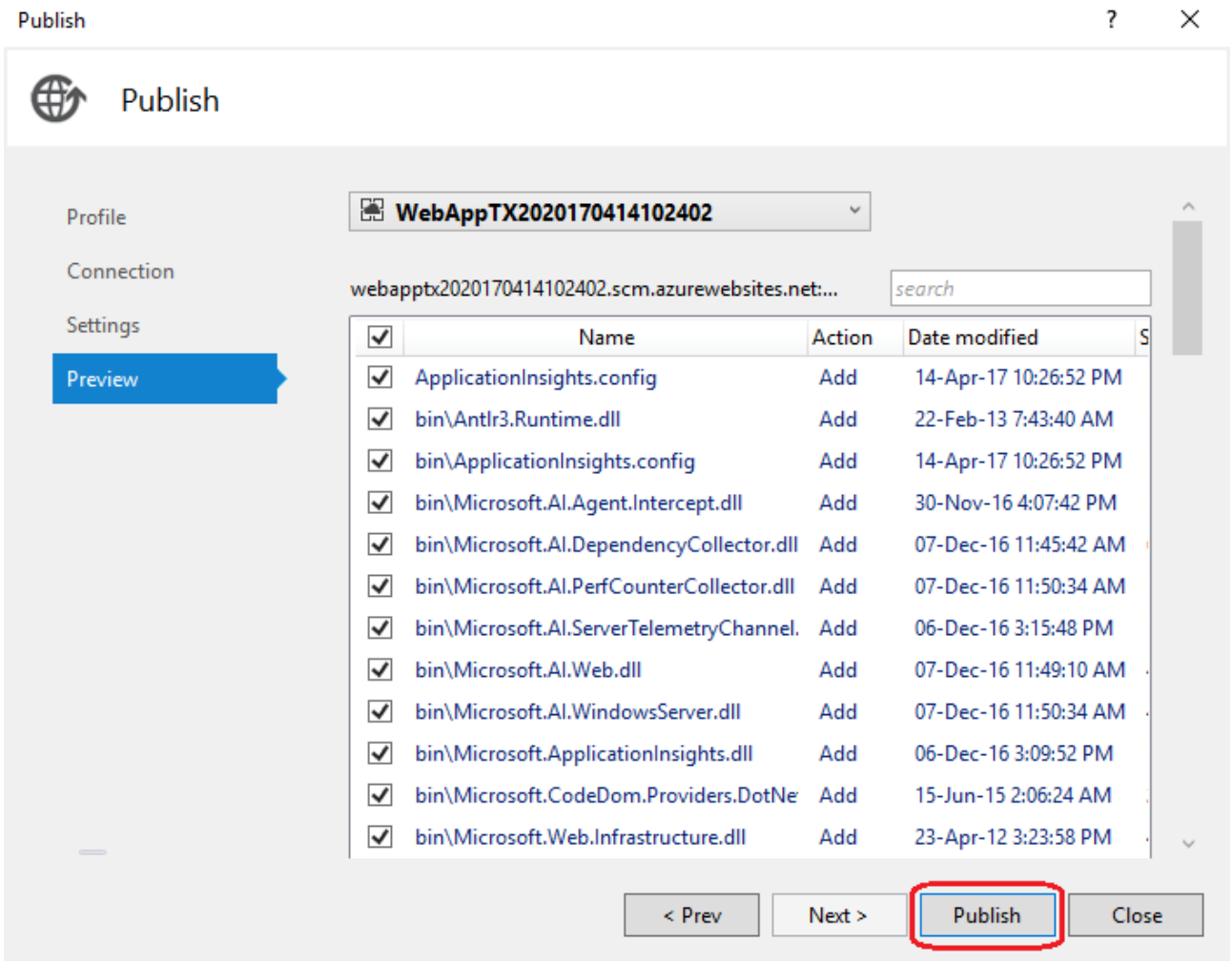


Fig.355

Il link da utilizzare come URL è:

http://webapptx2020170414102402.azurewebsites.net

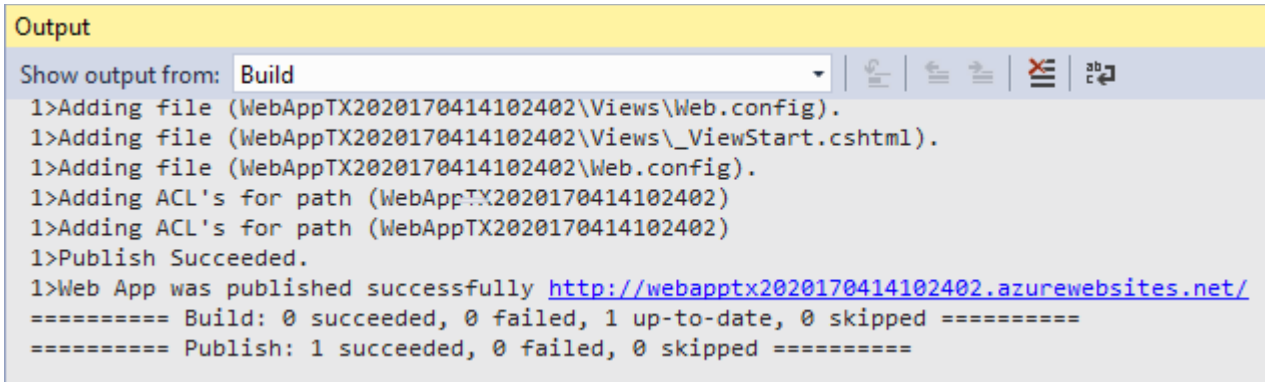


Fig.356

Aprire un qualsiasi browser inserendo il link pubblico della web app per verificarne il corretto funzionamento. In Fig.357 Google Chrome riporta la pagina di presentazione dell'applicazione web. Dal portale di Azure, si noterà che la pubblicazione ha prodotto la creazione di una voce di servizio chiamata WebAppTX2020170414102402 come da Fig.358 e Fig.359.

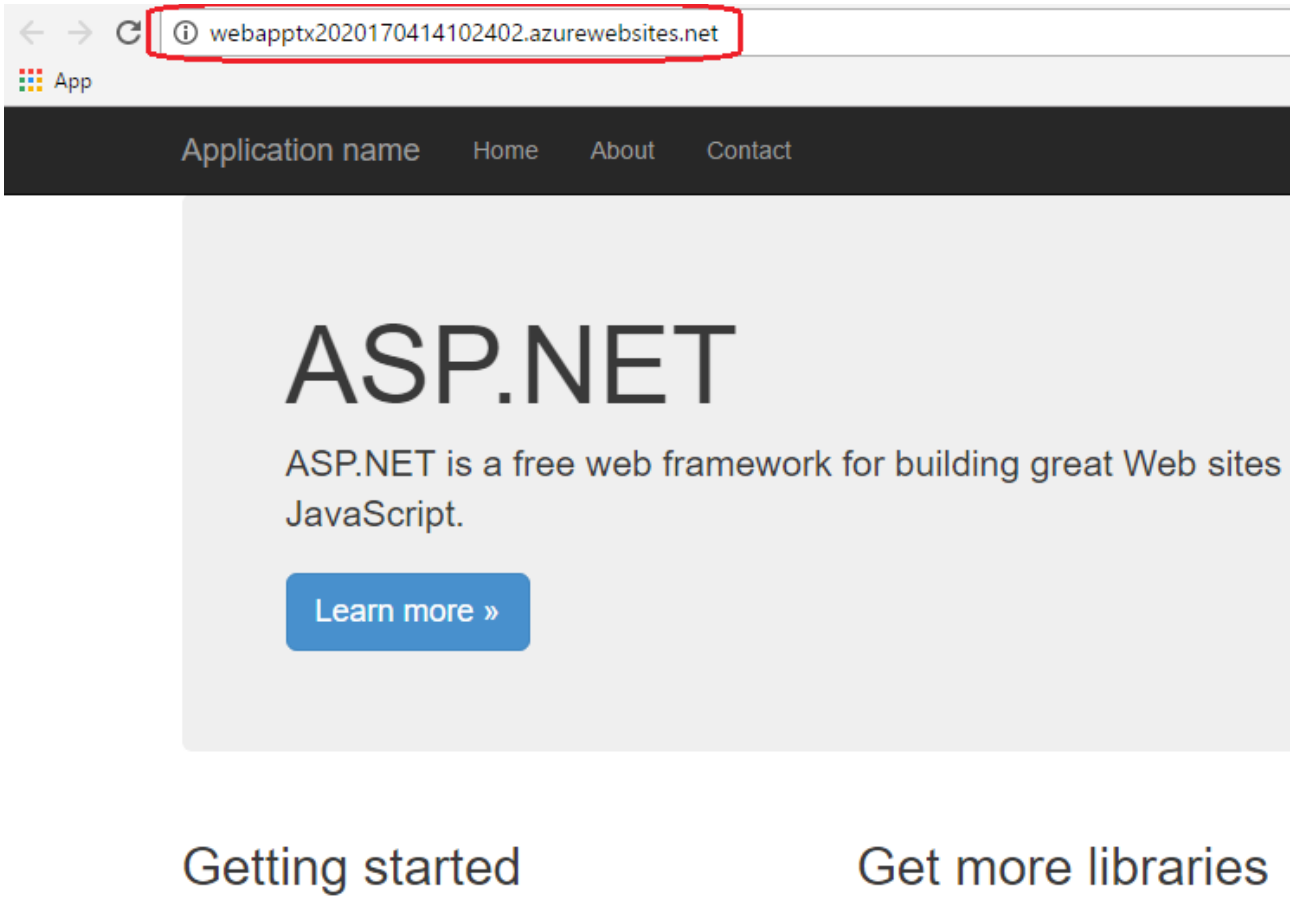


Fig.357

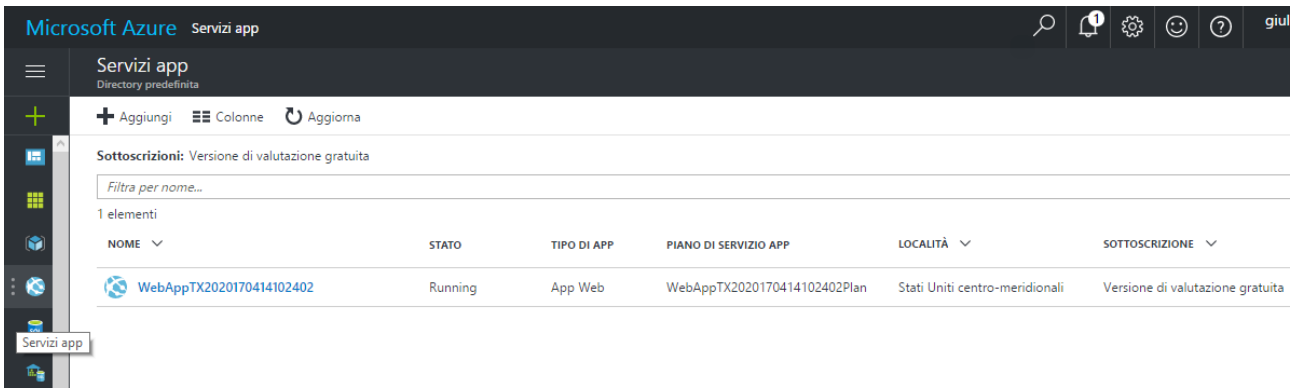


Fig.358

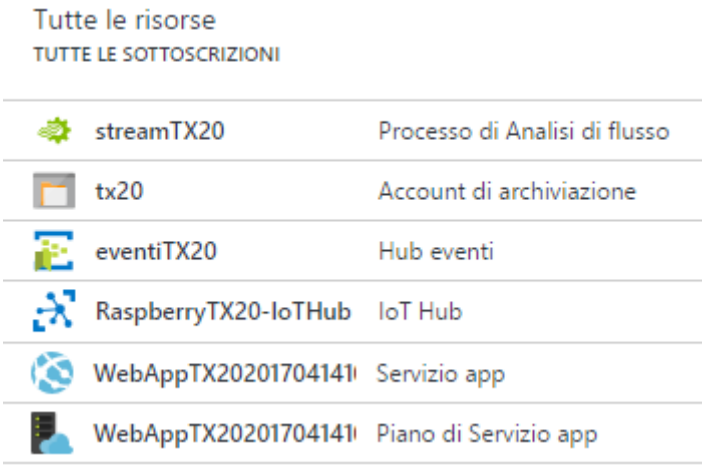


Fig.359

E' interessante notare che assieme al servizio WebAppTX2020170414102402, ne viene inserito un secondo di pianificazione, che permette di aumentare le prestazioni dell'applicazione web. E' possibile scegliere tra tre tipologie di tariffe, Basic, Standard e Premium, come si vede in Fig.360, oltre che a quella gratuita che non offre nessuna garanzia di backup e di prestazioni.

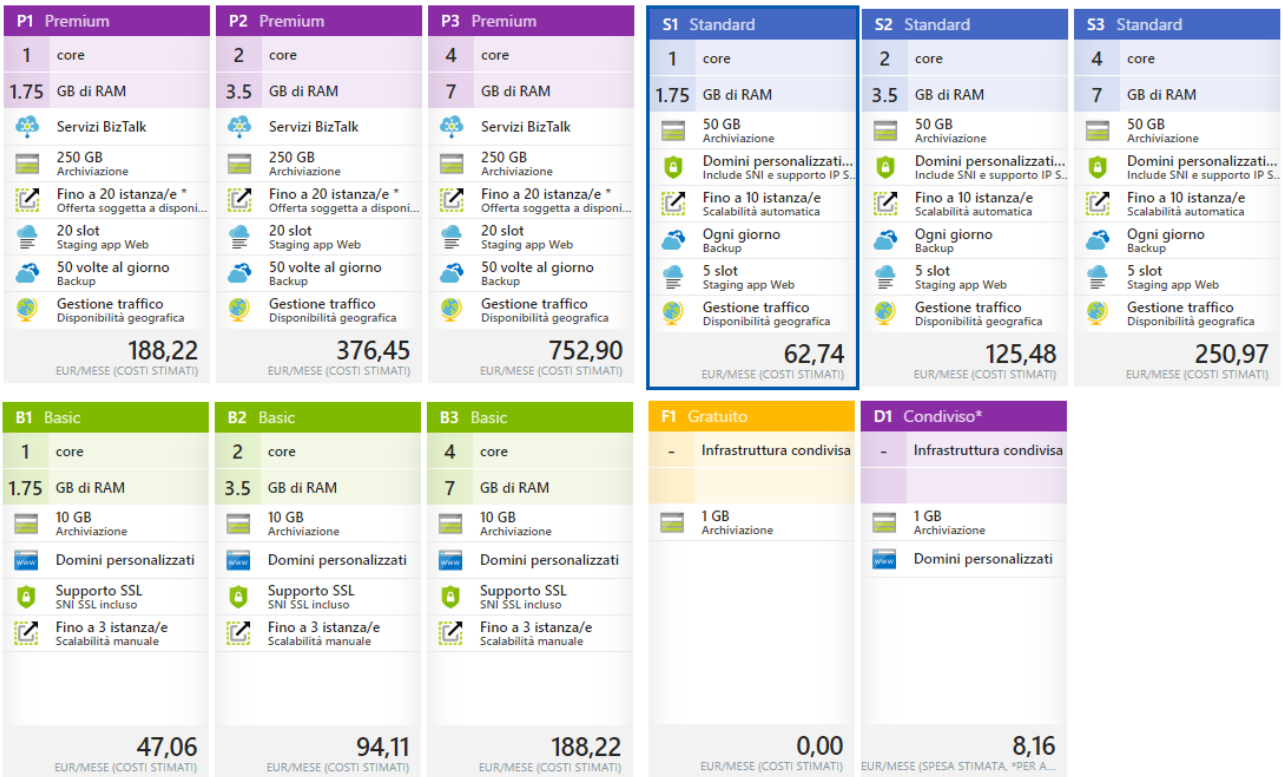


Fig.360

La fase di pubblicazione dell'applicazione è terminata, ma prima di iniziare a vedere la scrittura delle varie parti, è necessario installare la libreria "WindowsAzure.Storage" fondamentale per il prelievo dei dati dallo storage BLOB. L'installazione avviene sempre tramite il pacchetto "NuGet", come da Fig.361, installando l'ultima versione disponibile, la "8.1.1" al momento della stesura di questo paragrafo, accettando poi la licenza, Fig.362.

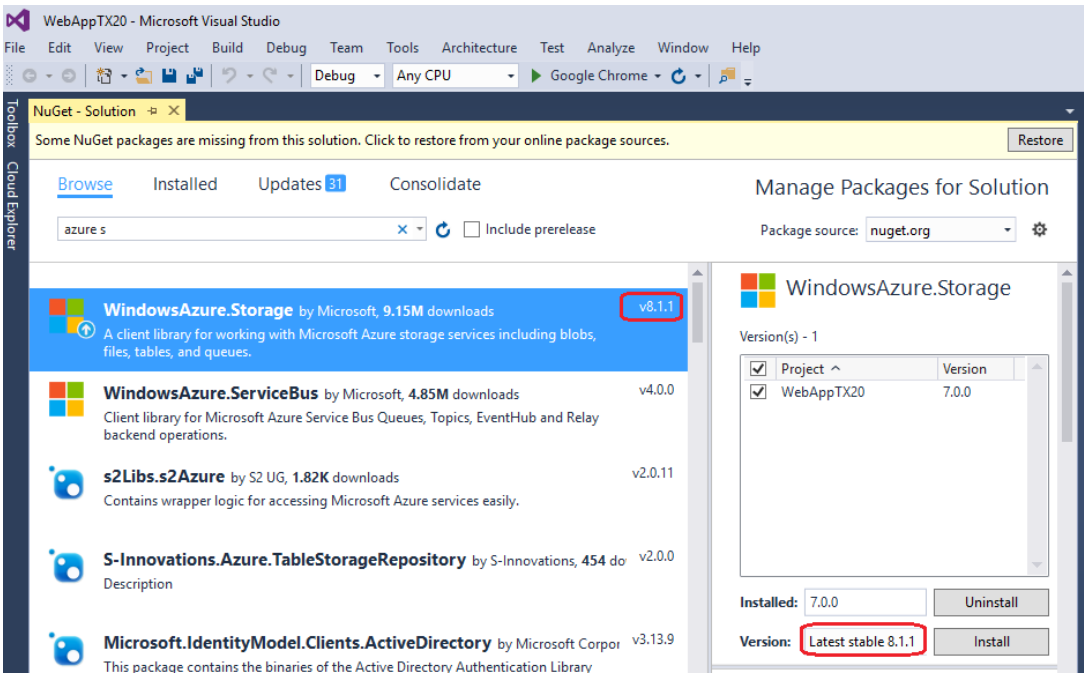


Fig.361

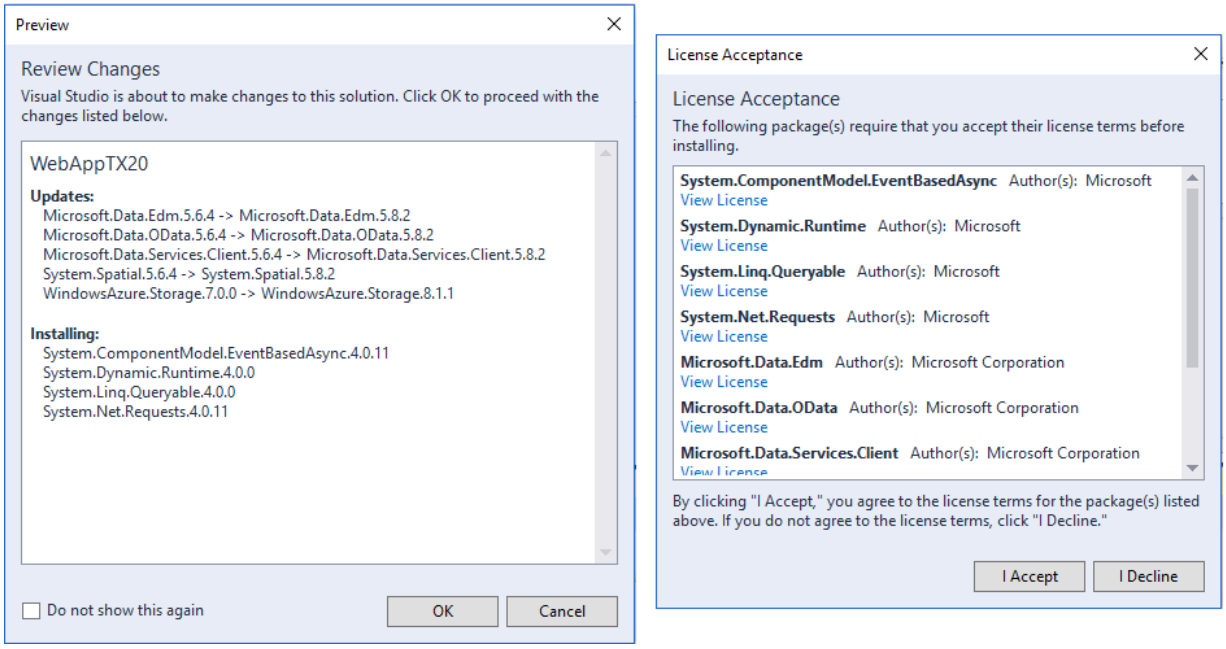


Fig.362

5.4.2.1 Models - TX20.cs

La realizzazione della web app segue il paradigma MVC, motivo per cui è fondamentale iniziare a scrivere sempre il modello, in tale caso saranno presenti alcune classi, tra cui TX20.cs che funge da contenitore dei dati una volta che sono stati recuperati dall'archivio. In questo contesto l'archivio non è un database, ma un file di testo con formato JSON recuperato dallo storage BLOB. Aggiungere la classe "TX20.cs" all'interno della directory "Models", come da Fig.363 e Fig.364.

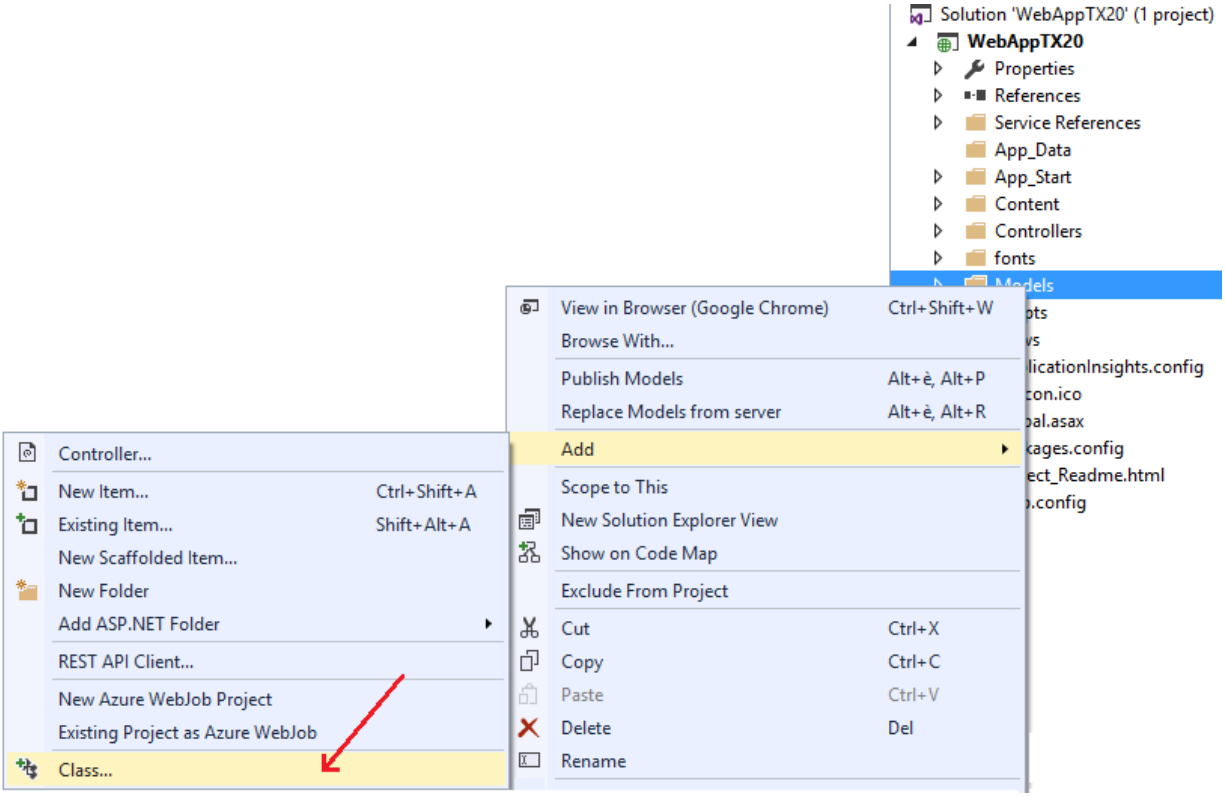


Fig.363

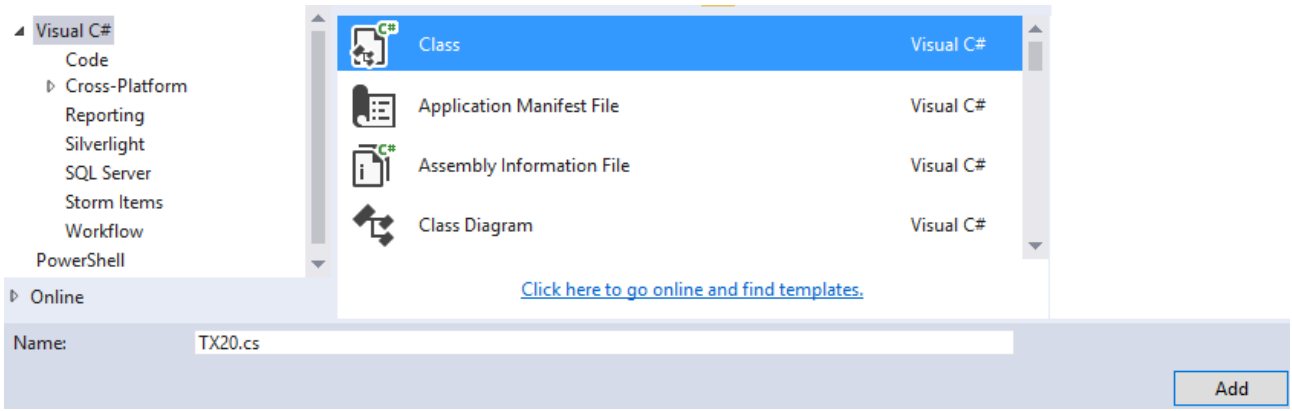


Fig.364

L'implementazione di questa classe è molto semplice, infatti è sufficiente definire i metodi per la memorizzazione e la lettura della singola informazione. I dati che si vogliono rendere disponibili lato web sono la direzione e velocità del vento, la data e l'ora della lettura, il tipo di vento (calma, brezza ecc...), la velocità massima giornaliera e l'ora in cui è stata rilevata, la velocità massima mensile ed annuale. L'applicazione farà uso di due grafici per rendere più interattiva l'interazione con l'utente, quindi viene utilizzato un vettore di tipo classe "DatiPerGrafico" che verrà poi impiegato nella vista per popolare i grafici con i dati. I metodi verranno impiegati come proprietà dell'oggetto TX20.

```
namespace WebAppTX20.Models
{
    public class TX20
    {
        public double TX20Speed { get; set; }
        public string TX20Direction { get; set; }
        public string Data { get; set; }
        public string Tempo { get; set; }
        public string TipoDiVento { get; set; }
        public double TX20SpeedMaxGiornaliera { get; set; }
        public string TempoTX20SpeedMax { get; set; }
        public double TX20SpeedMaxMensile { get; set; }
        public double TX20SpeedMaxAnnuale { get; set; }
        public DatiPerGrafico[] TX20ElencoParametriTX20 { get; set; }
    }
}
```

La classe "DatiPerGrafico" verrà creata all'intero della directory Models, ma nulla vieta di creare una cartella dedicata alle classi di appoggio con conseguente obbligo di impostare la clausola "using" per definire il percorso. Un esempio potrebbe essere la cartella "ClassiBase" con conseguente clausola:

```
using WebAppTX20.ClassiBase;
```

5.4.2.2 Models - TX20BusinessAccessLayer.cs

La parte logica di business chiamata "Business Access Layer", fa parte del blocco "Data Tier", e quindi del modello, e funge da intermediario tra il controllore e la classe contenitore TX20. Il codice è un pò lungo, ma non difficile. Per comodità si riporta tutto il listato al termine del paragrafo, analizzando in sequenza le varie parti. In Fig. 365 la classica creazione della classe, all'interno della directory "Models".

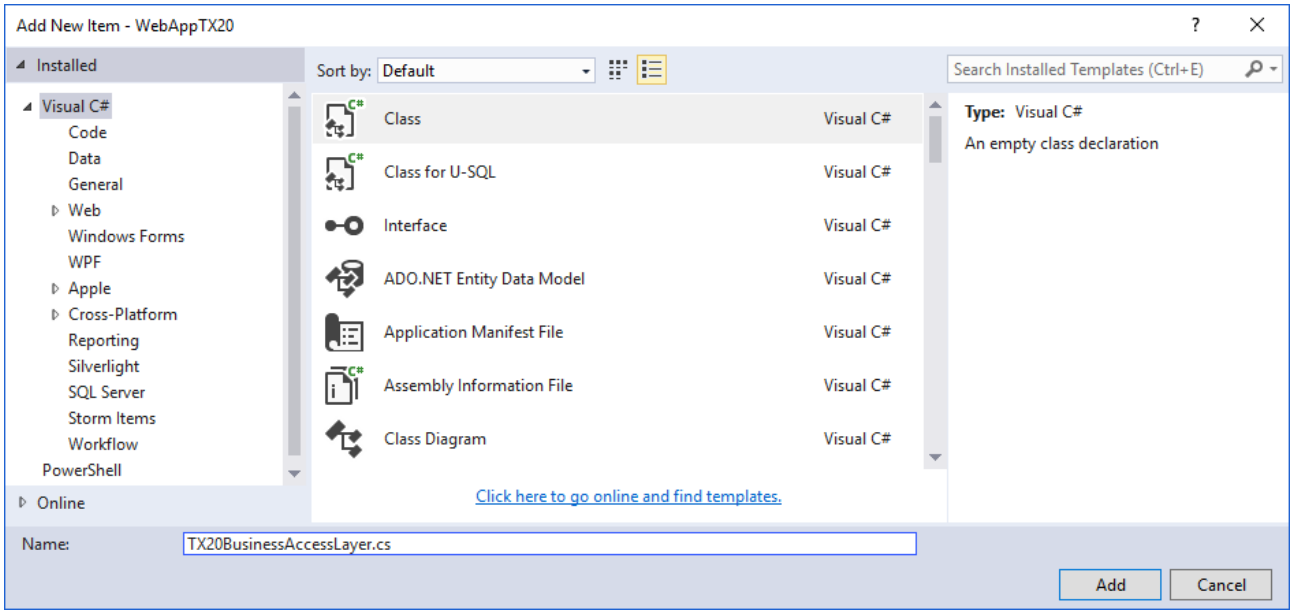


Fig.365

Il metodo cardine che verrà invocato dal controllore è "getDirectionSpeedDataTime()" che ha il compito di inizializzare un oggetto di tipo TX20 col fine di popolarlo con tutte le informazioni da veicolare alla vista. Il primo step è recuperare l'archivio con i dati direttamente dal cloud Azure, motivo per cui si fa uso di un metodo creato ad hoc "DownloadJSON" della classe "Azure", che provvede a salvare tutto il contenuto in formato JSON direttamente nella variabile di tipo stringa "json". Una volta in possesso di tutti i dati, si provvede ad un'operazione di parsing col fine di estrapolare le informazioni relative alla direzione, velocità, data e tempo di lettura. L'estrapolazione dei campi da una stringa è facilmente realizzabile utilizzando il metodo nativo "Split" nel quale si deve indicare il separatore di campo, che in tale caso è l'intera lettura con tutte le informazioni, motivo per cui la "Split" restituisce un vettore di stringhe. Per capire quale separatore utilizzare, è necessario vedere come vengono letti i dati dallo storage BLOB. Segue un esempio di estrapolazione dei dati.

{{"VelocitaMS":12.0,"Direzione":"NNE","Data":06-05-2017","Ora":"09-37-29"}} \r\n
{{"VelocitaMS":10.2,"Direzione":"NNE","Data":06-05-2017","Ora":"09-47-29"}} \r\n

Si nota la presenza dei caratteri '{ ... } \r\n' quali delimitatori della singola lettura, motivo per cui la sequenza di caratteri "}}\r\n" permettono di identificare una singola lettura completa dei relativi dati. Il vettore "sentences" ha tanti elementi quante sono le letture, quindi si dovrà passare in rassegna elemento per elemento dell'array, a cui corrisponde lettura dopo lettura, per procedere ad estrapolare i singoli campi sempre con la tecnica di splitting. Prima di procedere alla scansione si inizializzata una lista di tipo "DatiPerGrafico" che memorizzerà tutti i valori in modo che possano essere riportati nei due grafici presenti nella vista. La scansione del vettore prevede la manipolazione di una lettura che ha il formato precedente, ma senza i caratteri del precedente separatore.

{{"VelocitaMS":12.0,"Direzione":"NNE","Data":06-05-2017","Ora":"09-37-29"}

Da questa stringa si vuole applicare la "Split" in modo che estrapoli i campi quattro campi in un vettore chiamato "dati", motivo per cui si utilizza come separatore di campo un array di caratteri creato "on the fly" con i delimitatori '{' e '}' che permettono di ridurre la stringa ad una forma più contratta del tipo:

"VelocitaMS":12.0,"Direzione":"NNE","Data":06-05-2017","Ora":"09-37-29"

Il terzo carattere di questo vettore, ossia ',' permette di spaccare l'intera stringa nei quattro campi. Come opzione durante questa fase si decide di non includere nell'array eventuali stringhe vuote.

dati[0] ← "VelocitaMS":12.0,
dati[1] ← "Direzione":"NNE"
dati[2] ← "Data":06-05-2017"
dati[3] ← "Ora":"09-37-29"

Le singole informazioni non sono ancora però disponibili, motivo per cui i singoli campi dell'array "dati" subiranno un terzo ed ultimo meccanismo di splittamento, che permetta

ora di ottenere i dati nudi e crudi, con riferimento all'esempio i dati 12.0, "NNE", "06-05-2017" e "09-37-29". I quattro "Split" successivi utilizzano il carattere ":" come separatore di campo, provvedendo a modificare la data da "06-05-2017" a "06/05/2017" e l'ora da "09-37-29" a "09:37:29" sfruttando il metodo nativo "Replace" di tipo stringa.
Il passo finale consiste nel creare un oggetto "datiDelVento" che grazie alla relative proprietà, registrerà tutte le informazioni da inserire nella lista solo se le letture sono della giornata odierna, in caso contrario le proprietà saranno inizializzate con dati nulli. I grafici che verranno utilizzati nella vista riportano solamente dati relativi alle letture quotidiane. La velocità del vento viene convertita in numero reale e moltiplicata per lo scalare 3.6 così da avere il dato in Km/h con un numero di cifre decimali pari a 2. Per calcolare i gradi del vento da inserire nel diagramma della rosa dei venti, si utilizza il metodo "direzioneVentoInGradi()", mentre per stabilire il valore dell'ascissa X sul grafico delle velocità, viene utilizzato il metodo "calcolaStepXPerOra()" al quale va passato l'ora così che possa venire calcolato in modo corretto l'avanzamento X sul grafico. In seguito verrà ripreso tale metodo con maggiore dettaglio. Popolato l'oggetto "datiDelVento" non resta che inserirlo nella lista. Sempre all'interno del ciclo che scandisce tutte le letture del file JSON, resta da trovare la velocità massima giornaliera, mensile ed annuale utilizzando i metodi "controlloVelocitaGiornalieraMassima()", "controlloVelocitaMensileMassima()" e "controlloVelocitaMensileAnnuale()". Fuori dal ciclo iterativo si esegue l'azione finale di leggere l'ultima lettura della giornata, che verrà riportata in un'apposita textbox della pagina, motivo per cui si verifica se effettivamente esiste una lettura della giornata odierna, per procedere quindi all'inizializzazione dei campi tramite le proprietà dell'oggetto "InformazioniAnemometro" di tipo classe TX20. Le variabili che vengono utilizzate sono quelle inizializzate con gli ultimi valore del file JSON visto che essendo fuori dal ciclo l'ambito di visibilità di questi dati locali è ancora valido ed è pari all'ultima iterazione effettuata. In caso contrario i campi sono inizializzati con dati nulli o che hanno significato di assenza di letture odierne. L'ultima istruzione converte la lista di tipo "DatiPerGrafico" in un array e lo memorizza all'interno dell'oggetto "InformazioniAnemometro" come proprietà.
Il metodo "tipologiaDiVento()" restituisce il tipo di vento in base alla sua velocità. Nella Tab.18 è riportata la classificazione del tipo di vento.

Velocità vento [m/s]	Classificazione
fino a 1	calma
da 1 a 6	bava di vento
da 6 a 11	brezza leggera
da 11 a 19	brezza
da 19 a 28	brezza vivace
da 28 a 38	brezza tesa
da 38 a 49	vento fresca
da 49 a 61	vento forte
da 61 a 74	burrasca moderata
da 74 a 88	burrasca forte
da 88 a 102	tempesta
da 102 a 117	fortunale
oltre 117	uragano

Tab.18

Il metodo "controlloDataUltimaLetturaOdierna()" verifica che l'estrapolazione della data, relativa ad una lettura, sia quella della data odierna. La funzione utilizza il metodo "UtcNow" della classe statica "DateTime" aggiungendo due ore visto che l'esecuzione della web app sul server cloud riporta un ritardo di due ore. Il formato della data viene specificato secondo la localizzazione dell'Italia tramite la classe "CultureInfo". La data passata e quella estrapolata dal sistema vengono verificate nella clausola return tramite l'operatore ternario "?".

Il metodo "controlloVelocitaGiornalieraMassima()" estrapola, se presente, la massima velocità del giorno assieme alla data e ora. Questi dati vengono passati come riferimento al metodo che provvederà, in caso di presenza di una velocità maggiore del giorno odierno, ad aggiornare i valori di questi parametri formali. Per leggere dal sistema la data odierna, si impiega il metodo "DateTime.Today.Day" che in presenza di giorni da 1 a 9, restituisce una sola cifra, motivo per cui a livello di stringa viene concatenata con la stringa "0" così da potere venire correttamente comparato con la data estrapolata dalla lettura, per la quale la "Substring" riporta esclusivamente il giorno.

Il metodo "controlloVelocitaMensileMassima()" è realizzato sulla falsa riga del precedente, con lo scopo di rilevare nel mese corrente la velocità massima, ma senza riportare data e

ora di questa rilevazione. Vale il medesimo discorso per i mesi da 1 a 9, mentre per estrapolare dalla lettura il solo mese l'indice iniziale della "Substring" vale 3 per un lunghezza di 2 caratteri.
Il metodo "controlloVelocitaMensileAnnuale()" cerca la massima velocità dell'anno corrente, sempre seguendo la logica dei due metodi precedenti. L'anno viene estrapolato sempre grazie alla "Substring", partendo con un indice pari a 6 per una lunghezza di caratteri pari a 4.
Il metodo "direzioneVentoInGradi()" accetta come parametro in ingresso la direzione della lettura corrente e restituisce il valore in gradi da impiegare nel grafico della rosa dei venti.
L'ultimo metodo "calcolaStepXPerOra()" serve per determinare il valore sull'ascissa a cui posizionare le lettura. Per permettere un corretto avanzamento viene processata l'ora di lettura, comprensiva di minuti e secondi. In primis dall'ora si estrapolano i relativi campi sempre grazie all'impiego della "Substring", poi si applica un formula che restituisce un intero lungo. E' importante evidenziare che 1 secondo di tempo corrisponde allo scalare 1000, 1 minuto a 1000*60 ossia 60.000 mentre 1 ora a 1000*60*60 pari a 3.600.000. Questi tre dati scalari andranno moltiplicati per i rispettivi secondi, minuti ed ore. Nella formula complessiva è necessario considerare che alle ore "00:00:00" non corrisponde lo 0, ma bensì -3.600.000. Il motivo è legato all'implementazione del grafico Javascript di cui si parlerà in seguito. Qualora si dimenticasse tale correttivo, il punto verrebbe sempre graficato in modo corretto, ma una volta che l'utente si posizionerebbe sul punto, il codice Javascript riporterebbe l'ora sfalsata in avanti di 60 minuti. Segue la formula finale.

X = 1000*(secondiLettura) + 60.000*(minutiLettura) + 3.600.000*(oraLettura) - 3.600.000

Segue il codice completo della classe "TX20BusinessAccessLayer.cs".

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Globalization;

namespace WebAppTX20.Models
{
```

```
public class TX20BusinessAccessLayer
{
   public TX20 getDirectionSpeedDataTime()
   {
      string ContainerName = "dati";
      string BlobName = "0_65a5b5bb2e8b418f8faa4fa5f876c49b_1.json";

      TX20 InformazioniAnemometro = new TX20();

      string json = new Azure().DownloadJSON(ContainerName, BlobName);

      string[] stringSeparators = new string[] { "}}\r\n" };

      string[] sentences = json.Split(stringSeparators, StringSplitOptions.None);

      string data = String.Empty;
      string tempo = String.Empty;
      string[] speedInformations = { String.Empty };
      string directionWithOutDoubleQuote = String.Empty;
      string dataWithOutDoubleQuote = String.Empty;
      string oraWithOutDoubleQuote = String.Empty;
      string dataGiornalieraSpeedMax = String.Empty;
      string tempoGiornalieroSpeedMax = String.Empty;
      double velocitaMassimaGiornaliera = 0;
      double velocitaMassimaMensile = 0;
      double velocitaMassimaAnnuale = 0;
      bool presenzaUltimaLetturaDataOdierna;

      List<DatiPerGrafico> elencoDatiPerGrafico = new List<DatiPerGrafico>();

      for (int idx = 0; idx < sentences.Length; idx++)
      {
        var dati = sentences[idx].Split(new[] { '{', '}', ',' },
```

```
                                        StringSplitOptions.RemoveEmptyEntries);

    string speedMSRaw = dati[0];
    string directionRaw = dati[1];
    string dataRaw = dati[2];
    string oraRaw = dati 3];

    speedInformations = speedMSRaw.Split(':');

    string[] directionInformations = directionRaw.Split(':');
    directionWithOutDoubleQuote = directionInformations[1].Substring(1,
                                        directionInformations[1].Length - 2);

    string[] dateInformations = dataRaw.Split(':');
    dataWithOutDoubleQuote = dateInformations[1].Substring(1,
                                        dateInformations[1].Length - 2);
    string[] oraInformations = oraRaw.Split(':');
    oraWithOutDoubleQuote = oraInformations[1].Substring(1,
                                          oraInformations[1].Length - 2);

    dataWithOutDoubleQuote = dataWithOutDoubleQuote.Replace('-', '/');
    oraWithOutDoubleQuote = oraWithOutDoubleQuote.Replace('-', ':');

    DatiPerGrafico datiSulVento = new DatiPerGrafico();

   presenzaUltimaLetturaDataOdierna = controlloDataUltimaLetturaOdierna
                                        (dataWithOutDoubleQuote);

    if (presenzaUltimaLetturaDataOdierna)
    {
       datiSulVento.velocitaVento = Math.Round(
                                Convert.ToDouble(speedInformations[1]) * 3.6, 2);
       datiSulVento.direzioneVento = directionWithOutDoubleQuote;
```

```
            datiSulVento.direzioneVentoInGradi = direzioneVentoInGradi(
                                                    directionWithOutDoubleQuote);
            datiSulVento.conversioneOraPerStepX = calcolaStepXPerOra(
                                                   oraWithOutDoubleQuote);

            elencoDatiPerGrafico.Add(datiSulVento);
        }

        controlloVelocitaGiornalieraMassima(dataWithOutDoubleQuote,
                                            oraWithOutDoubleQuote,
                                            speedInformations[1],
                                            ref velocitaMassimaGiornaliera,
                                            ref dataGiornalieraSpeedMax,
                                            ref tempoGiornalieroSpeedMax);

        controlloVelocitaMensileMassima(dataWithOutDoubleQuote,
                                        oraWithOutDoubleQuote,
                                        speedInformations[1],
                                        ref velocitaMassimaMensile);

        controlloVelocitaMensileAnnuale(dataWithOutDoubleQuote,
                                        oraWithOutDoubleQuote,
                                        speedInformations[1],
                                        ref velocitaMassimaAnnuale);
    }

    presenzaUltimaLetturaDataOdierna = controlloDataUltimaLetturaOdierna(
                                            dataWithOutDoubleQuote);

    if (presenzaUltimaLetturaDataOdierna)
    {
        InformazioniAnemometro.TX20Direction = directionWithOutDoubleQuote;
```

```
            InformazioniAnemometro.TX20Speed = Math.Round(
                                    Convert.ToDouble(speedInformations[1]) * 3.6, 2);
            InformazioniAnemometro.Data = dataWithOutDoubleQuote;
            InformazioniAnemometro.Tempo = oraWithOutDoubleQuote;
            InformazioniAnemometro.TipoDiVento = tipologiaDiVento(
                                        InformazioniAnemometro.TX20Speed);
            InformazioniAnemometro.TX20SpeedMaxGiornaliera = Math.Round(
                                        velocitaMassimaGiornaliera * 3.6, 2);
            InformazioniAnemometro.TempoTX20SpeedMax = tempoGiornalieroSpeedMax;
            InformazioniAnemometro.TX20SpeedMaxMensile = Math.Round(
                                        velocitaMassimaMensile * 3.6, 2);
            InformazioniAnemometro.TX20SpeedMaxAnnuale = Math.Round(
                                        velocitaMassimaAnnuale * 3.6, 2);
        }
        else
        {
            InformazioniAnemometro.TX20Direction = "-";
            InformazioniAnemometro.TX20Speed = 0;
            InformazioniAnemometro.Data = "-";
            InformazioniAnemometro.Tempo = "-";
            InformazioniAnemometro.TipoDiVento = "-";
            InformazioniAnemometro.TX20SpeedMaxGiornaliera =
                                                velocitaMassimaGiornaliera * 3.6;
            InformazioniAnemometro.TempoTX20SpeedMax = "-";
            InformazioniAnemometro.TX20SpeedMaxMensile =
                                                velocitaMassimaMensile * 3.6;
            InformazioniAnemometro.TX20SpeedMaxAnnuale =
                                                velocitaMassimaAnnuale * 3.6;
        }

        InformazioniAnemometro.TX20ElencoParametriTX20 =
                                    elencoDatiPerGrafico.ToArray<DatiPerGrafico>();
```

```
    return InformazioniAnemometro;
}

private string tipologiaDiVento(double tx20Speed)
{
    if (tx20Speed < 1)
        return "calma";

    if (tx20Speed >= 1 && tx20Speed < 6)
        return "bava di vento";

    if (tx20Speed >= 6 && tx20Speed < 11)
        return "brezza leggera";

    if (tx20Speed >= 11 && tx20Speed < 19)
        return "brezza";
    if (tx20Speed >= 19 && tx20Speed < 28)
        return "brezza vivace";

    if (tx20Speed >= 28 && tx20Speed < 38)
        return "brezza tesa";

    if (tx20Speed >= 38 && tx20Speed < 49)
        return "vento fresca";

    if (tx20Speed >= 49 && tx20Speed < 61)
        return "vento forte";

    if (tx20Speed >= 61 && tx20Speed < 74)
        return "burrasca moderata";

    if (tx20Speed >= 74 && tx20Speed < 88)
        return "burrasca forte";
```

```
        if (tx20Speed >= 88 && tx20Speed < 102)
            return "tempesta";

        if (tx20Speed >= 102 && tx20Speed < 117)
            return "fortunale";
        else
            return "uragano";
    }

    private bool controlloDataUltimaLetturaOdierna(string dataUltimaLettura)
    {
        string data = System.DateTime.UtcNow.AddHours(2).ToString(
                                                        "d", new CultureInfo("it-IT"));

        return data.Equals(dataUltimaLettura) ? true : false;
    }

    private void controlloVelocitaGiornalieraMassima(
                string data, string tempo, string letturaVelocitaStr,
                ref double velocitaMassima, ref string dataVelMax, ref string tempoVelMax)
    {
        double letturaVelocita = Convert.ToDouble(letturaVelocitaStr);

        string giornoOdierno = data.Substring(0, 2);

        string oggi = String.Empty;

        if (DateTime.Today.Day.ToString().Length == 1)
            oggi = "0" + DateTime.Today.Day.ToString();
        else
            oggi = DateTime.Today.Day.ToString();

        if (giornoOdierno.Equals(oggi) && letturaVelocita >= velocitaMassima)
```

```
    {
        velocitaMassima = letturaVelocita;
        dataVelMax = data;
        tempoVelMax = tempo;
    }
}

private void controlloVelocitaMensileMassima(string data, string tempo, string
                                    letturaVelocitaStr, ref double velocitaMassimaMensile)
{
    double letturaVelocita = Convert.ToDouble(letturaVelocitaStr);

    string meseOdierno = data.Substring(3, 2);

    string mese = String.Empty;
    if (DateTime.Today.Month.ToString().Length == 1)
        mese = "0" + DateTime.Today.Month.ToString();
    else
        mese = DateTime.Today.Month.ToString();

    if (meseOdierno.Equals(mese) && letturaVelocita >= velocitaMassimaMensile)
    {
        velocitaMassimaMensile = letturaVelocita;
    }
}

private void controlloVelocitaMensileAnnuale(string data, string tempo, string
                                    letturaVelocitaStr, ref double velocitaMassimaAnnuale)
{
    double letturaVelocita = Convert.ToDouble(letturaVelocitaStr);

    string annoOdierno = data.Substring(6, 4);
```

```
        string anno = DateTime.Today.Year.ToString();

        if (annoOdierno.Equals(anno) && letturaVelocita >= velocitaMassimaAnnuale)
        {
            velocitaMassimaAnnuale = letturaVelocita;
        }
    }

    private double direzioneVentoInGradi(string direzioneVento)
    {
        double direzioneInGradi = 0;

        switch (direzioneVento)
        {
            case "N":
                direzioneInGradi = 0.0;
            break;

            case "NNE":
                direzioneInGradi = 22.5;
             break;

            case "NE":
                direzioneInGradi = 45.0;
             break;

            case "ENE":
                direzioneInGradi = 67.5;
              break;

            case "E":
                direzioneInGradi = 90.0;
             break;
```

```
case "ESE":
   direzioneInGradi = 112.5;
 break;

case "SE":
   direzioneInGradi = 135.0;
 break;

case "SSE":
   direzioneInGradi = 157.5;
 break;

case "S":
   direzioneInGradi = 180.0;
break;

case "SSW":
   direzioneInGradi = 202.5;
break;

case "SW":
   direzioneInGradi = 225.0;
 break;

case "WSW":
   direzioneInGradi = 247.5;
 break;

case "W":
   direzioneInGradi = 270.0;
 break;

case "WNW":
```

```
                direzioneInGradi = 292.5;
                break;

            case "NW":
                direzioneInGradi = 315.0;
                break;

            case "NNW":
                direzioneInGradi = 337.5;
                break;
        }
        return direzioneInGradi;
    }

    private long calcolaStepXPerOra(string oraDiRilevazioneVento)
    {
        string ora = oraDiRilevazioneVento.Substring(0, 2);
        string minuti = oraDiRilevazioneVento.Substring(3, 2);
        string secondi = oraDiRilevazioneVento.Substring(6, 2);

        // 1 secondo è pari ad uno step sul grafico di 1000
        // 1 minuti è pari ad uno step sul grafico di 60.000
        // 1 ora è pari ad uno step sul grafico di 60*60000 = 3.600.000

        long stepSecondiX = 1000 * Convert.ToInt32(secondi);
        long stepMinutiX = 60000 * Convert.ToInt32(minuti);
        long stepOraX = 3600000 * Convert.ToInt32(ora);
        long stepX = stepOraX + stepMinutiX + stepSecondiX - 3600000;

        return stepX;
    }
  }
}
```

5.4.2.3 Azure.cs

La creazione della classe avviene sempre all'interno della directory "Models", come da Fig. 366.

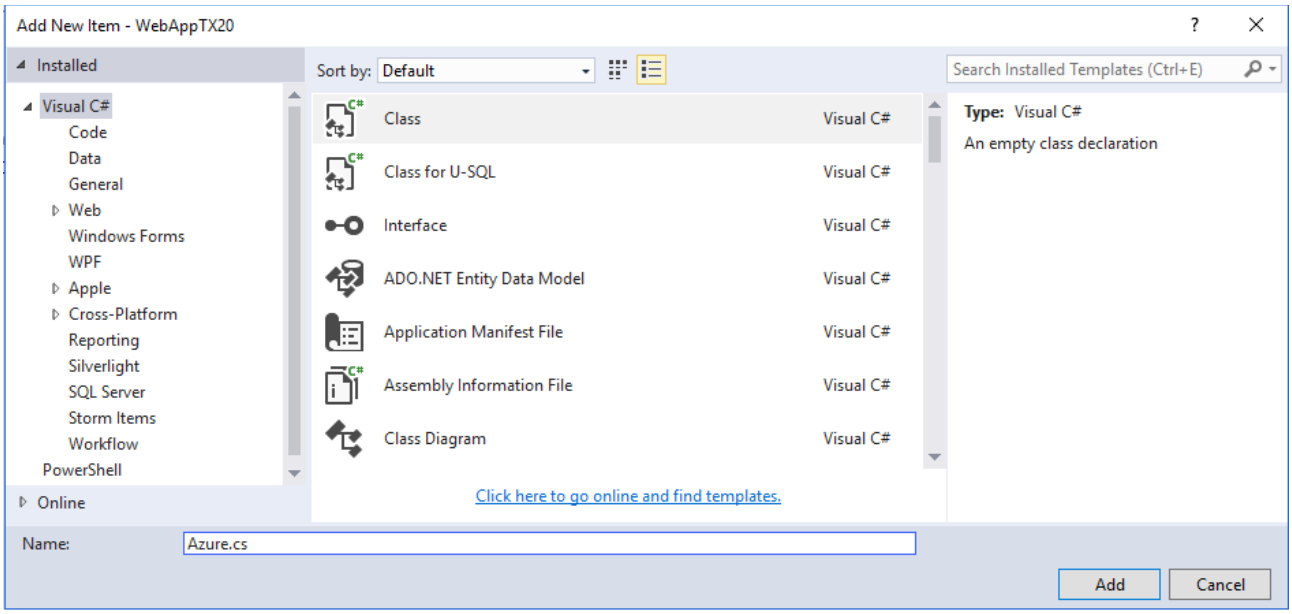


Fig.366

Questa classe offre un solo metodo pubblico "DownloadJSON()" che serve per scaricare il file testo in formato JSON dallo storage BLOB del cloud Microsoft. Questa funzione richiama il metodo privato "GetContainer()" che restituisce il container "dati" dell'account configurato nel file Web.config, con il tag "connAzure". Successivamente dal container "dati" si provvede a scaricare il file in loco con estensione ".json", restituendolo alla classe modello per la fase di splitting esposta nel paragrafo precedente. Tutte le classi statiche "CloudBlobClient", "CloudBlobContainer", "ConfigurationManager" e "CloudStorageAccount" fanno parte del pacchetto "WindowsAzure.Storage", motivo per cui è necessario utilizzare l'apposita clausola "using", previa installazione del pacchetto, che però è stata effettuata dopo la creazione del progetto. Segue il listato della classe.

```
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;
using System.Configuration;
```

```
namespace WebAppTX20.Models
{
    public class Azure
    {
        public Azure() { }

        private CloudBlobContainer GetContainer(string ContainerName)
        {
            CloudBlobClient clientBLOB;
            CloudBlobContainer containterBLOB;

            var connectionString =
                  ConfigurationManager.ConnectionStrings["connAzure"].ConnectionString;

            var storageAccount = CloudStorageAccount.Parse(connectionString);

            clientBLOB = storageAccount.CreateCloudBlobClient();
            containterBLOB = clientBLOB.GetContainerReference(ContainerName);

            return containterBLOB;
        }

        public string DownloadJSON(string ContainerName, string BlobName)
        {
            CloudBlobContainer blobContainer = GetContainer(ContainerName);
            CloudBlockBlob blockBlob = blobContainer.GetBlockBlobReference(BlobName);

            blockBlob.FetchAttributes();

            return blockBlob.DownloadText();
        }
    }
}
```

5.4.2.4 DatiPerGrafico.cs

La creazione della classe avviene sempre all'interno della directory "Models", come da Fig. 367.

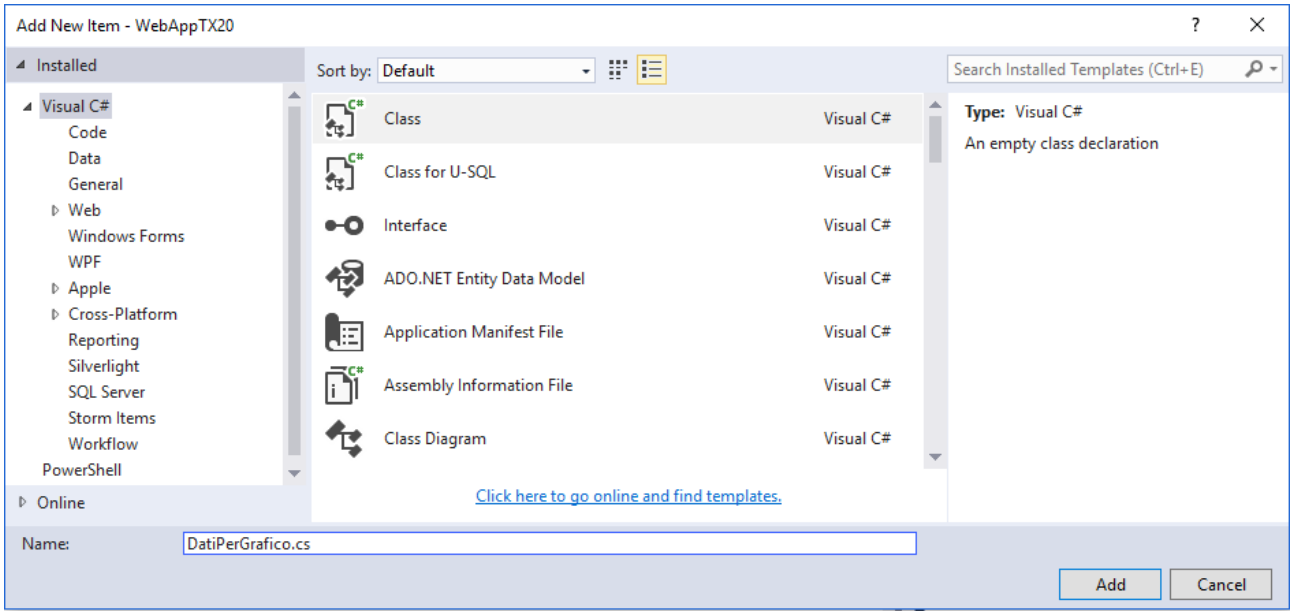


Fig.367

Questa classe serve per memorizzare i dati della singola lettura col fine di graficare l'informazione nella pagina web. A livello di modello ogni oggetto di questa classe viene salvato in una lista, così da poter graficare un pool di letture. I metodi getter e setter permettono la memorizzazione dei dati come proprietà.

```
namespace WebAppTX20.Models
{
    public class DatiPerGrafico
    {
        public double velocitaVento { get; set; }
        public double direzioneVentoInGradi { get; set; }
        public string direzioneVento { get; set; }
        public long conversioneOraPerStepX { get; set; }
    }
}
```

5.4.2.5 Controller - MeteoController.cs

La creazione della classe avviene all'interno della directory "Controllers", per cui sopra tale directory premere il tasto destro del mouse e procedere all'inserimento del controller tramite la voce "Add ►Controller...". Si veda la Fig. 368.

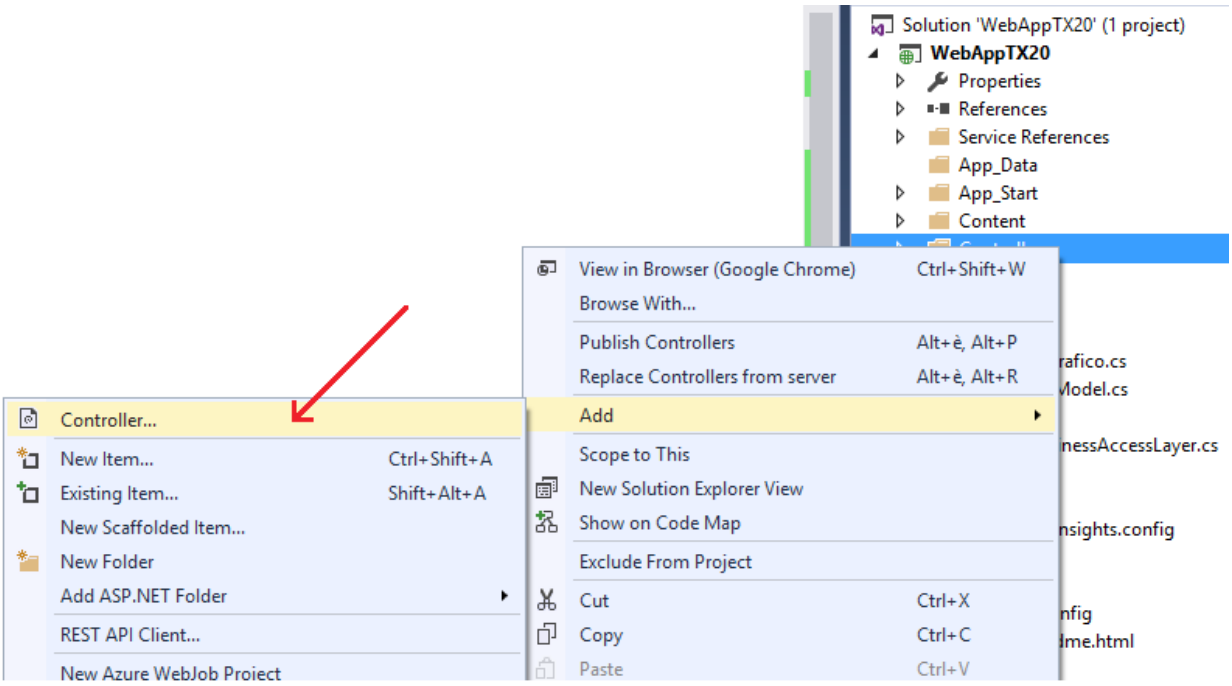


Fig.368

Selezionare la voce "MVC 5 Controller - Empty" e premere "Add", come da Fig.369.

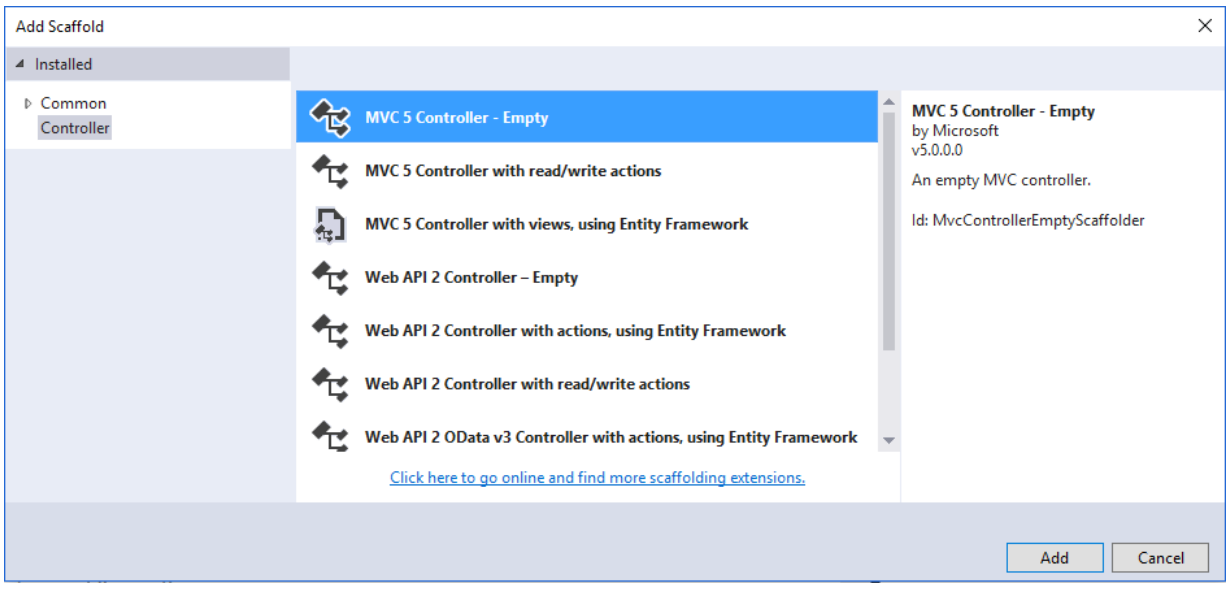


Fig.369

Indicare il nome del controllore in "MeteoController" come da Fig.370.

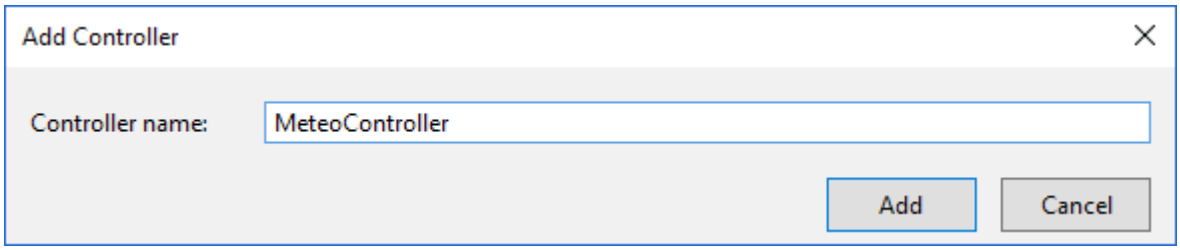


Fig.370

Questa classe controllore raccoglie la richiesta del client e restituisce, sotto forma di azione, una pagina web che rappresenta la vista. La classe "MeteoController" deve estendere la classe "Controller". Il metodo "TX20RivaDelGarda()" crea un oggetto di Business Access Layer e, grazie alla "getDirectionSpeedDataTime()", vengono letti i dati e memorizzati in una variabile di tipo "TX20", la quale contiene anche tutte le informazioni da graficare nella pagina web. La vista potrà interagire con i dati grazie alla definizione di un parametro personalizzato chiamato a scelta "DatiTX20" che rappresenta l'intero contenuto della variabile "ParametriAnemometroTX20" di tipo TX20. Detto in altro modo è un parametro che permette di visualizzare sulla vista i dati prelevati dal modello. Il metodo restituisce la vista chiamata "index" senza indicare l'estensione ".cshtml" e passando il contenuto del parametro "DatiTX20". E' possibile testare in locale il metodo "TX20RivaDelGarda()" utilizzando il path "//Meteo/TX20RivaDelGarda" perché venga correttamente chiamato.

```
using System.Web.Mvc;
using WebAppTX20.Models;

namespace WebAppTX20.Controllers
{
    public class MeteoController : Controller
    {
        // http://localhost:<socket>//Meteo/TX20RivaDelGarda
        public ActionResult TX20RivaDelGarda()
        {
```

```
            TX20BusinessAccessLayer Bal = new TX20BusinessAccessLayer();
            TX20 ParametriAnemometroTX20 = Bal.getDirectionSpeedDataTime();
            ViewData["DatiTX20"] = ParametriAnemometroTX20;

            return View("index");
        }
    }
}
```

5.4.2.6 Views - index.cs.html

Il progetto utilizza una sola e semplice vista per visualizzare i dati prelevati dal modello. Il lettore noterà che la web app ha provveduto automaticamente ha creare la directory "Views ► Meteo", quindi si deve creare una vista in questa cartella sempre tramite il tasto destro mouse e le voce "Add ► View..." come da Fig.371.

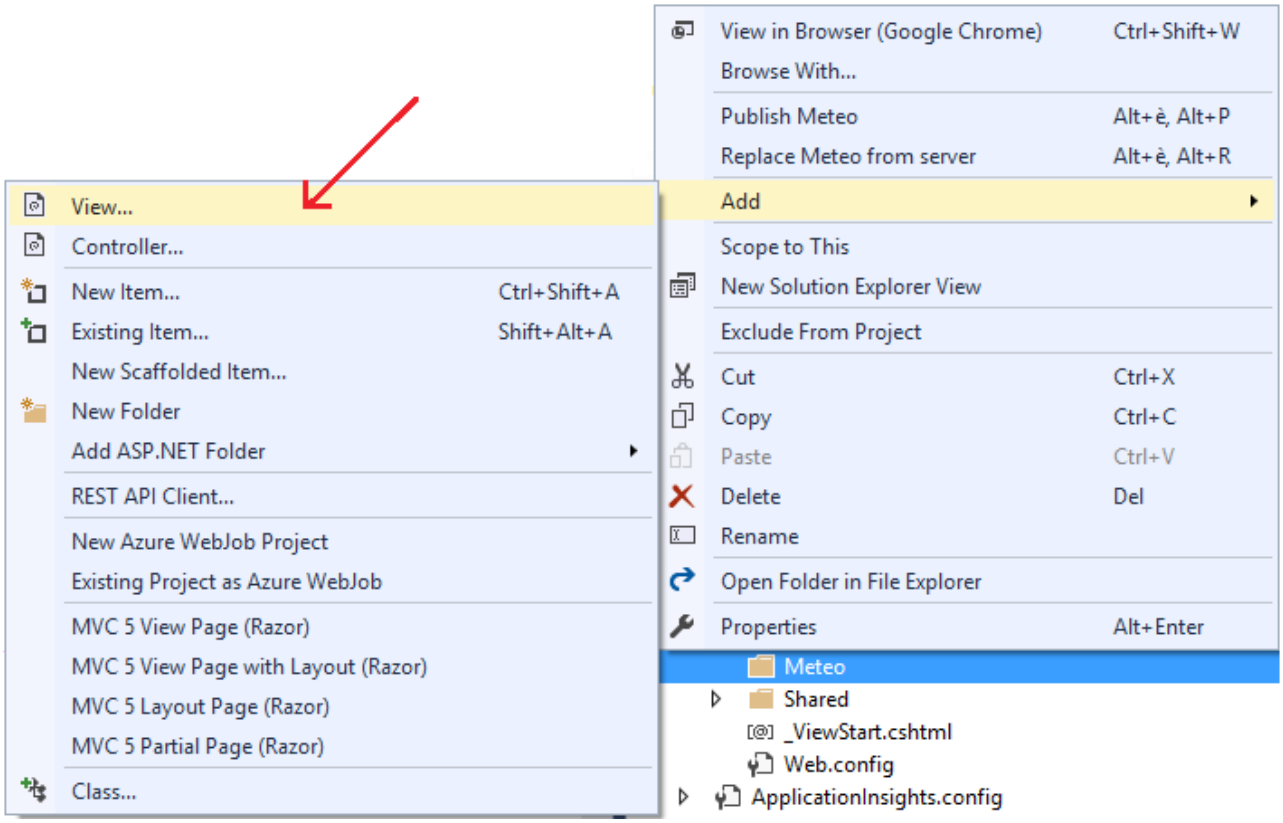


Fig.371

Chiamare la vista "index" togliendo la spunta "Use a layout page:". L'estensione ".cs.html" verrà aggiunta automaticamente. Si veda la Fig.372.

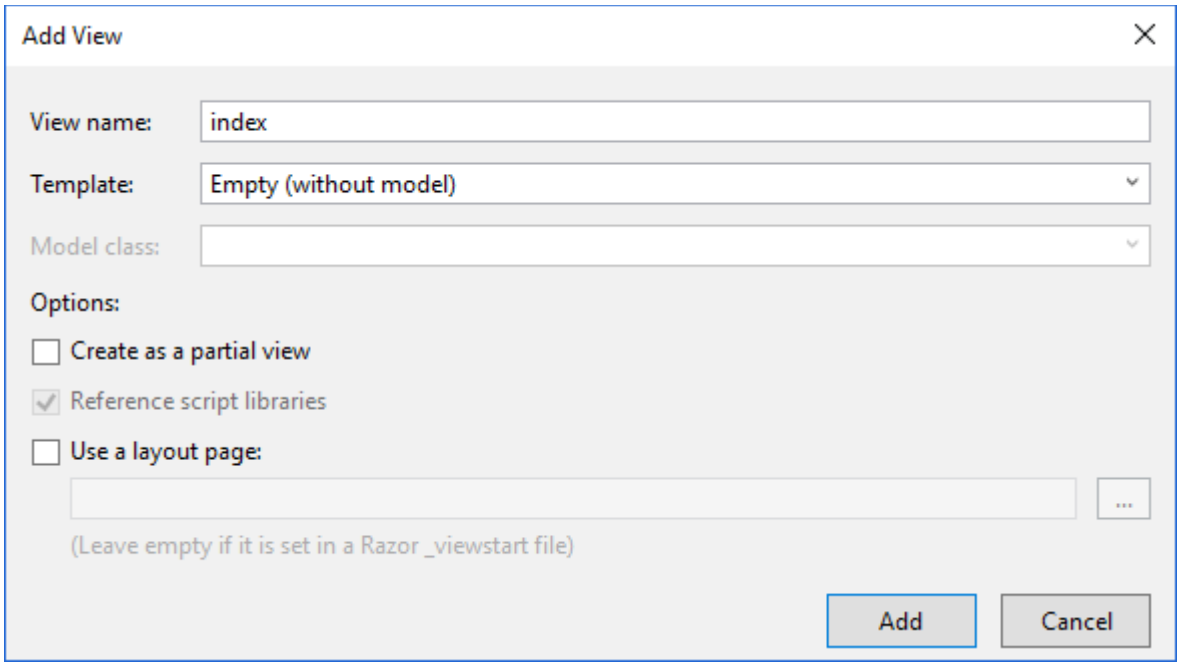


Fig.372

La vista "index.cs.html" combina l'uso dei tag HTML, fogli di stile CSS, codice Javascript e ASP.NET Razor C#, quindi è necessario avere una conoscenza minima in ognuno di questi argomenti, perché una revisione da zero delle singole tematiche richiederebbe centinaia e centinaia di pagine. Il codice completo della vista segue in fondo al paragrafo, e si provvede a descrivere le parti salienti, senza entrare nella descrizione dei singoli files Javascript. La vista riporta in sintesi le informazioni dell'ultima lettura, indicando la direzione e velocità del vento in km/h, il tipo di vento e la massima velocità giornaliera con l'ora in cui è stata rilevata. In un'apposita sezione sono riportate la velocità massima giornaliera, mensile ed annuale senza nessuna indicazione su data ed ora di rilevamento. Seguono poi due semplici grafici Javascript, il primo mostra l'andamento temporale delle letture in termini di velocità del vento, e posizionandosi col mouse sopra ogni punto graficato è possibile conoscere la direzione vento e l'ora di rilevamento. Il secondo grafico è una rosa dei venti, che posiziona i punti in base alla direzione della massa d'aria, permettendo all'utente di auto scalare il grafico sulla base delle velocità delle singole letture. Posizionandosi col mouse sopra ogni punto è possibile conoscere la direzione in termini di gradi. In Fig.373 e Fig.374 la realizzazione finale della vista.

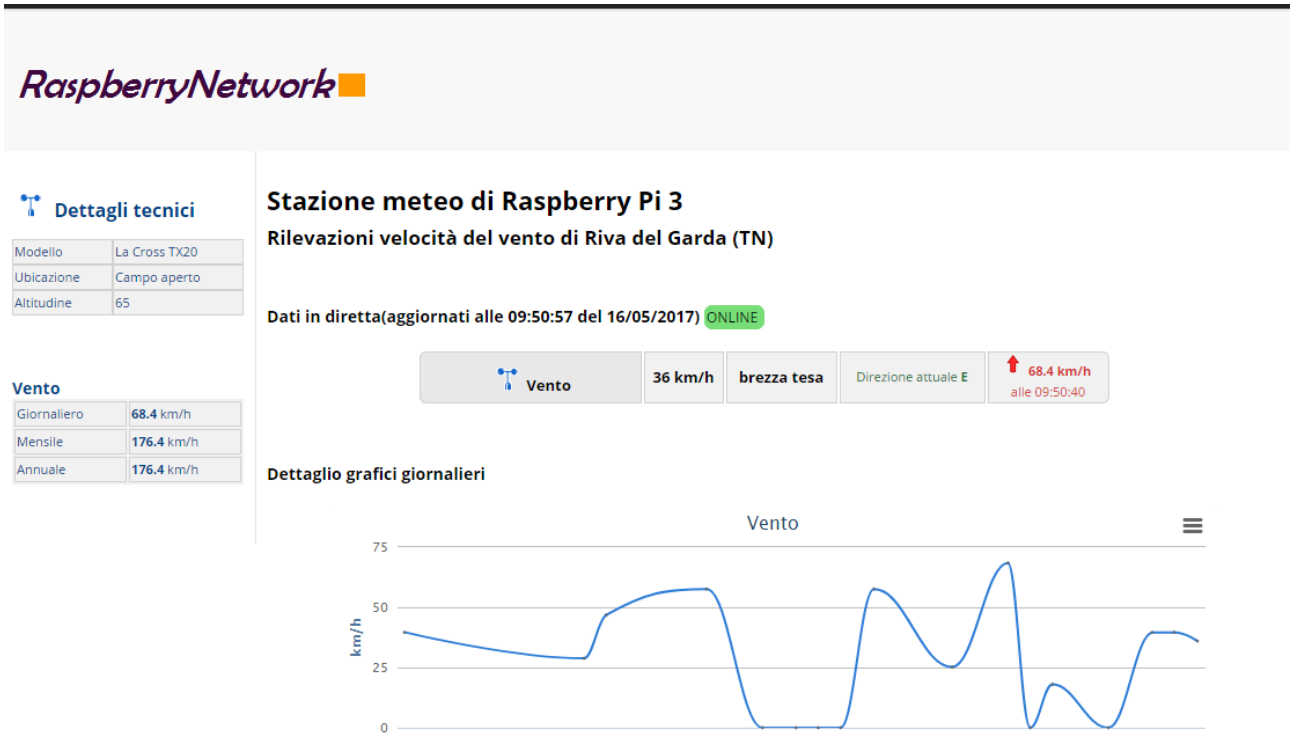


Fig.373

Cliccando sul pulsante "Vento" di Fig.374 è possibile riscalare i punti in base alla velocità.

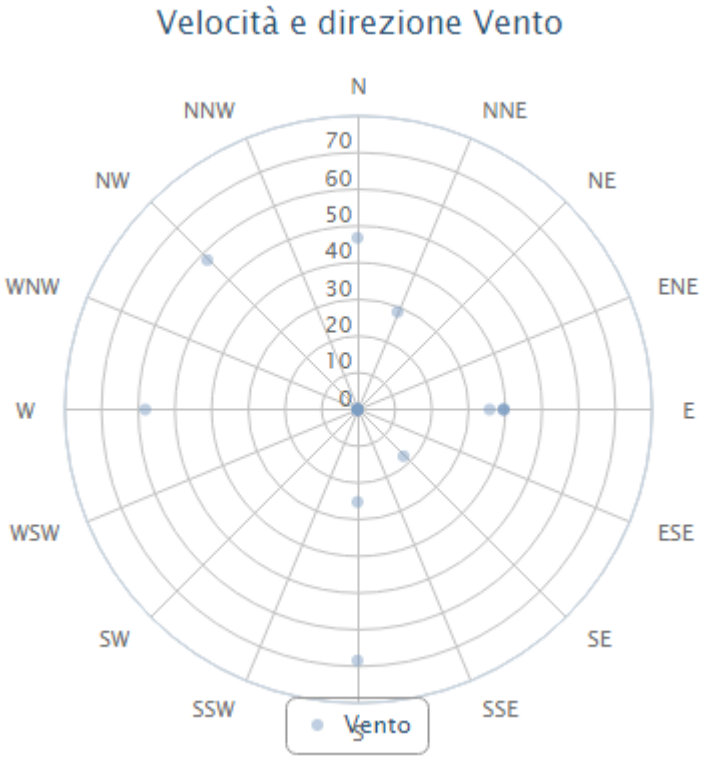


Fig.374

La prima sezione di codice HTML aggancia tra i tag <HEAD> </HEAD> tutti i Javascript necessari ed i fogli di stile CSS alla vista, files che poi dovranno venire copiati nelle rispettive directory "Content" e "Scripts".
Segue un estratto del codice finale.

```
<head profile="http://www.w3.org/1999/xhtml/vocab">
   <title>Stazione meteo Raspberry Pi 3</title>
      <link href="~/Content/style21b26.css?v2" rel="stylesheet" type="text/css" />
       ....
      <script type="text/javascript" src="~/Scripts/jquery.min.js"></script>
       ....
</head>
```

La pagina web è stata formatta con una prima sezione di navigazione vuota utilizzando il tag DIV chiamato "navigation-mnw" ed assegnando all'interno altri DIV con i rispettivi CSS. Per capire in modo semplice la logica di progettazione, viene utilizzato Adobe Dreamweaver in modalità Split così da evidenziare contemporaneamente codice e design. La Fig.375 riporta la sezione di navigazione vuota.

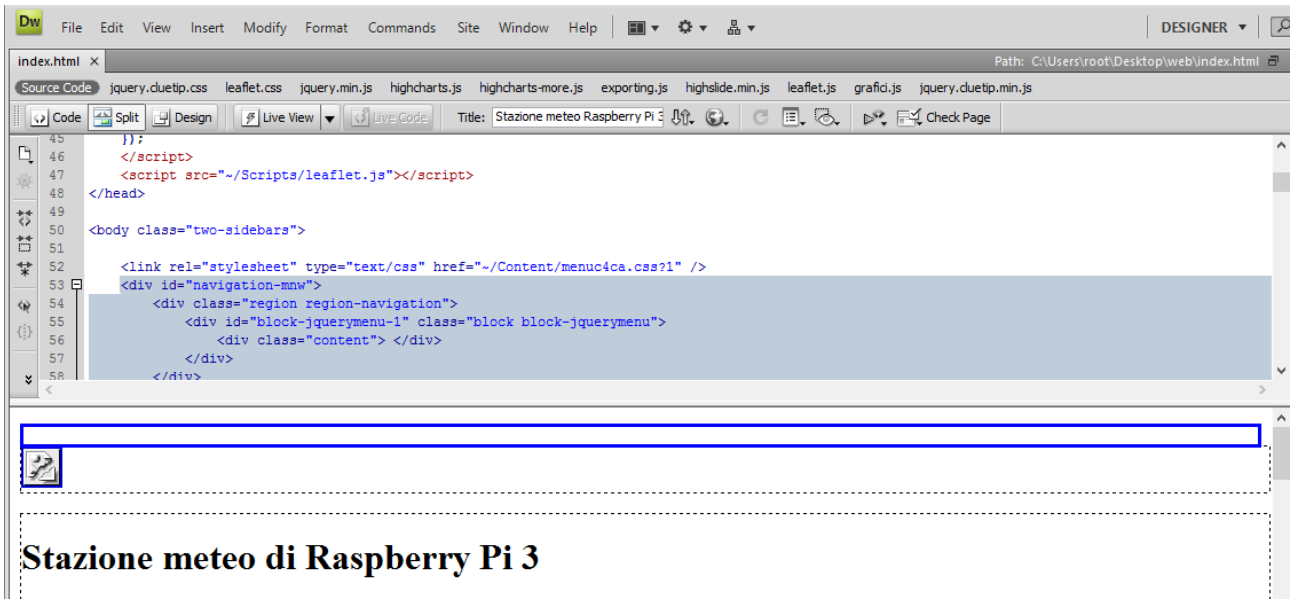


Fig.375

La seconda sezione DIV è l'intestazione, dove verrà inserito un logo personalizzato chiamato "logo_header.png" da inserire successivamente nell'apposita directory "Content/Images". Si veda la Fig.376.

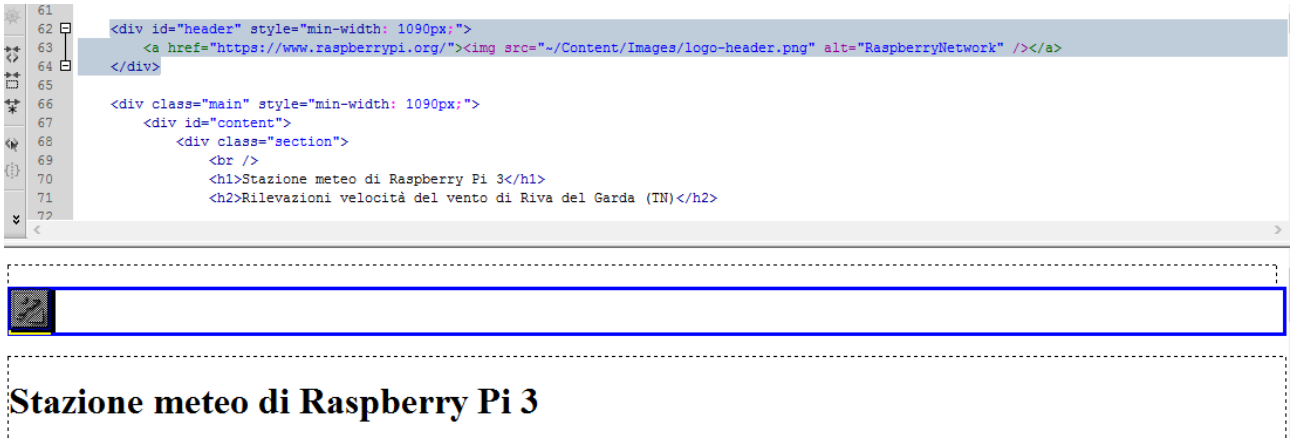


Fig.376

Il terzo DIV racchiude tutto il corpo della pagina, quindi tutte le informazioni testuali e i grafici. Si veda la Fig.377.

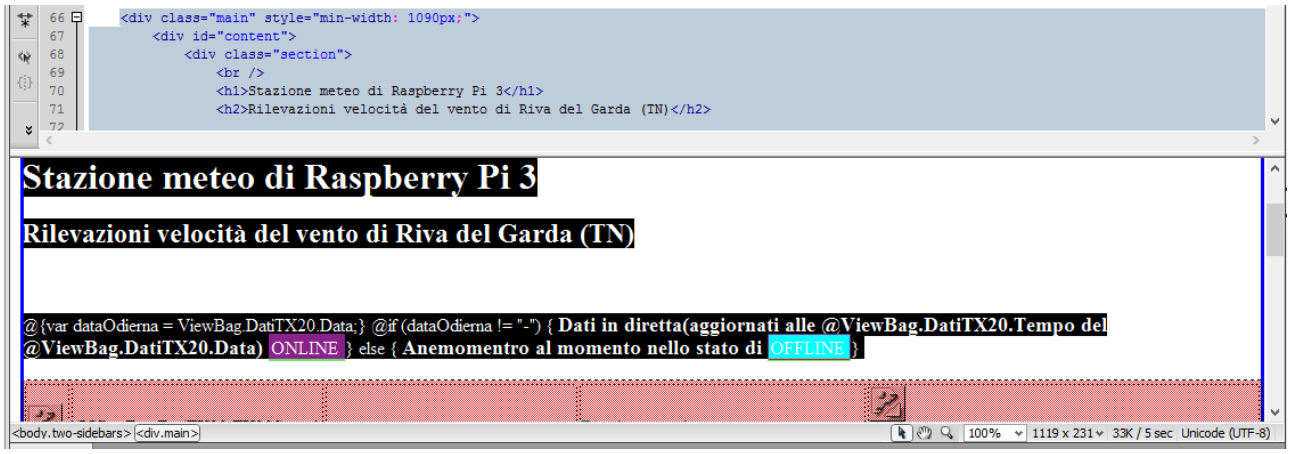


Fig.377

Il quarto ed ultimo DIV definisce il footer della pagina dove vengono riportate delle semplici informazioni di importanza secondaria, come riferimenti societari, dati del progettista, informazioni sulla privacy e/o altri dettagli. La Fig.378 evidenzia il codice ed il design del footer.

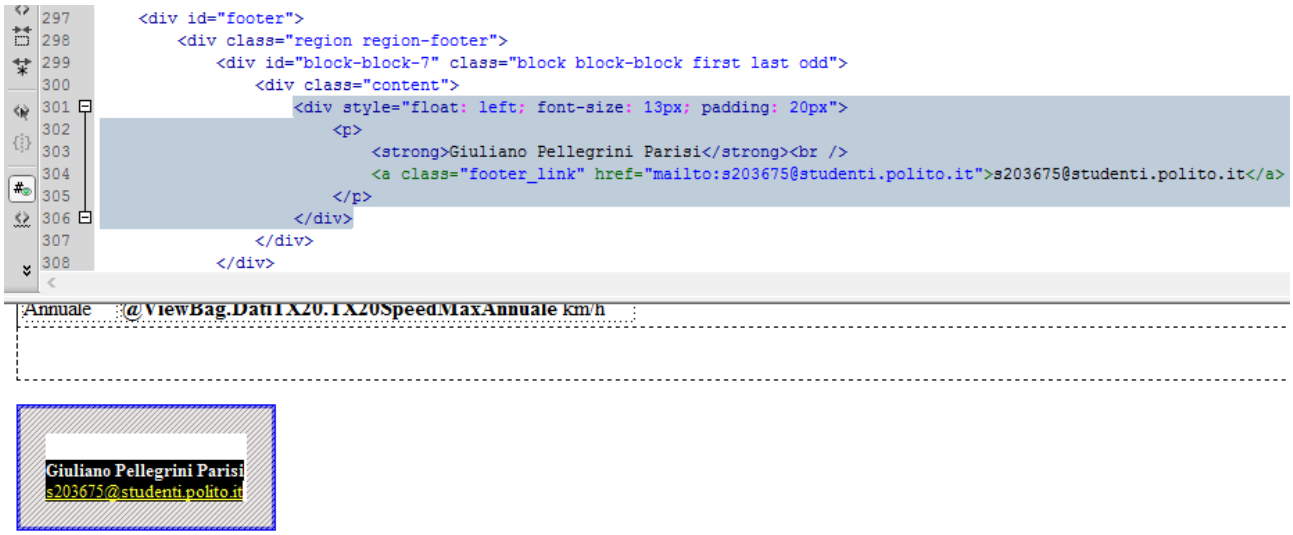


Fig.378

Le informazioni testuali relative all'ultima lettura e alla massima velocità giornaliera, sono visualizzate nella sezione centrale della pagina. Ovviamente potrebbe accadere che i dati rilevati dall'anemometro non arrivino allo storage cloud, ad esempio nel caso non sia disponibile una connettività di rete sul Pi 3. In tale circostanza la pagina dovrà fornire un indicazione che nessuna lettura odierna è disponibile. Per fare questo controllo è necessario ricordare che il controllore "MeteoController.cs" passa alla pagina web i dati prelevati dal modello, assegnandoli il nome "DatiTX20". Sotto il codice del metodo "TX20RivaDelGarda()".

```
TX20BusinessAccessLayer Bal = new TX20BusinessAccessLayer();
TX20 ParametriAnemometroTX20 = Bal.getDirectionSpeedDataTime();

ViewData["DatiTX20"] = ParametriAnemometroTX20;
```

La variabile "ParametriAnemometroTX20" di tipo classe "TX20" contiene tutti i dati della singola lettura, con l'aggiunta della massima lettura quotidiana, mensile ed annuale e tutte le singole letture da graficare. Tutte queste informazioni sono passate alla pagina web sotto il nome "DatiTX20". Per accedere alle singole informazioni, è necessario usare le proprietà della classe TX20, ossia i metodi getter/setter che sono stati definiti. Per accedere ad esempio alla velocità dell'ultima lettura dalla vista, basta digitare la seguente

stringa "DatiTX20.TX20Speed". Tale scrittura però lega il codice HTML al codice C#, quindi l'attore che permette di implementare tale connubio si chiama Razor. La sezione del codice Razor inizia sempre con il carattere '@', permettendo poi di scrivere il codice all'interno delle parentesi graffe. Segue il classico esempio di una sezione Razor.

```
@{
    // mio codice
}
```

Per definire una variabile chiamata "dataOdierna" inizializzata con la velocità dell'ultima lettura basta definire il seguente codice Razor.

```
@{
    var dataOdierna = ViewBag.DatiTX20.Data;
}
```

Come si nota viene utilizzata la proprietà "ViewBag" per accedere ai dati inseriti lato controllore dall'altra proprietà "ViewData". Entrambe le proprietà sono state costruite appositamente per lavorare con il paradigma MVC. Detto in altre parole, la "ViewData" inietta i dati verso Razor, mentre la "ViewBag" preleva i dati da Razor. Qualora si voglia leggere il contenuto di una proprietà della classe TX20 in Razor, senza utilizzare variabili, è possibile accedere in modo rapido nel seguente modo.

```
@ViewBag.DatiTX20.TX20Speed ;
```

E' possibile definire un controllo condizionale Razor con @if{ }, oppure un ciclo iterativo con @For, quindi è chiaro che si hanno a disposizioni i classici costrutti di un linguaggio di programmazione. La trattazione completa di questo argomento esula dallo scopo di questa esposizione, ma serve esclusivamente avere delle nozioni di base per leggere i dati inviati dal controllore. Il primo blocco di codice Razor della pagina effettua un controllo sulla presenza o meno di una lettura della giornata odierna, visto che anomalie di rete potrebbero isolare il Raspberry con l'ovvia impossibilità di registrare i dati nel cloud Azure. Il controllo della lettura odierna avviene con un blocco condizionale Razor @if{ }.

La variabile "dataOdierna" ha memorizzato la data del giorno, oppure la stringa "-" qualora la lettura dei dati dallo storage BLOB non siano del medesimo giorno. Questa strategia è stata già esposta nel paragrafo 5.4.2.2. Il codice Razor si mescola ai tag HTML così da permettere la visualizzazione delle informazioni, infatti il codice sottostante qualora sia presente una lettura del giorno, stampa ora e data, così da indicare all'utente che l'anemometro TX20 è online. In caso contrario, blocco "else", si riporta un messaggio che avverte che il dispositivo è offline. Ogni tipologia di pagina web dinamica ha i propri meccanismi che permettono l'interazione dei dati con la parte di design HTML.

```
@{var dataOdierna = ViewBag.DatiTX20.Data;}

        @if (dataOdierna != "-")
        {
            <h3 style="display: inline;">
                Dati in diretta(aggiornati alle @ViewBag.DatiTX20.Tempo del
                                @ViewBag.DatiTX20.Data)
            </h3>
                <span style="display: inline;border: 1px solid #77DD77;background-color:
                             #77DD77;border-radius: 7px;padding: 1px;">
                    ONLINE
                </span>
        }
        else
        {
            <h3 style="display: inline;">
                Anemometro al momento nello stato di
            </h3>
                <span style="display: inline;border: 1px solid #77DD77;background-color:
                             #FF0000;border-radius: 7px;padding: 1px;">
                    OFFLINE
                </span>
        }
```

Un secondo pezzo di codice che vale la pena analizzare, riguarda la visualizzazione dei dati in tabella nella parte centrale del corpo della pagina. Il tutto in una sola riga. I dati da riportare riguardano esclusivamente l'ultima lettura della giornata odierna, accompagnata dalla massima velocità del giorno. Qualora non sia presente nessuna informazione giornaliera, non viene visualizzato nulla in griglia. Il controllo è il medesimo del caso precedente, infatti viene sempre testata la variabile "dataOdierna". L'accesso ai dati avviene sempre con Razor sfruttando le proprietà della classe TX20, come spiegato in precedenza. Si visualizza la velocità e direzione del vento, il tipo di vento e la massima giornaliera con l'ora di rilevamento. Segue il blocco di istruzioni.

```
<table>
  ...
  <col />
  <tbody>
    <tr>
        @if (dataOdierna != "-")
        {
            <td class="headr"><img style="margin: -10px 0 0 0;"
                src="~/Content/Images/wind.png" alt="Vento" />Vento
             </td>
             <td class="subhead"><span>@ViewBag.DatiTX20.TX20Speed km/h
                                    </span><br />
             </td>
             <td class="subhead"><span>@ViewBag.DatiTX20.TipoDiVento
                                     </span><br />
             </td>
             <td style="color:#287233">Direzione attuale
                          <strong>@ViewBag.DatiTX20.TX20Direction
                          </strong>
             </td>
             <td style="color:#D53032"><img src="~/Content/Images/up2.png"
                  alt="" /> <strong>@ViewBag.DatiTX20.TX20SpeedMaxGiornaliera km/h
                  </strong> <br />alle @ViewBag.DatiTX20.TempoTX20SpeedMax
```

```
                </td>
                }
            </tr>
        </tbody>
</table>
```

Un altro blocco di codice interessante è la visualizzazione sulla sinistra della pagina della massima velocità giornaliera, mensile ed annuale, senza però indicare data ed ora. Il blocco di codice che realizza questa parte è simile al precedente, si accede direttamente al valore da stampare a video tramite le proprietà della classe TX20, come "TX20SpeedMaxGiornaliera", "TX20SpeedMaxMensile" e "TX20SpeedMaxAnnuale". L'inserimento di questi dati avviene all'interno di una tabella HTML.

```
<div class="box3">
    <div class="left_b">
        <span class="w"><b>Vento </b> </span><br />
        <table cellpadding="1" class="toolsidebar">
            <tr><td>Giornaliero</td><td><b>@ViewBag.DatiTX20.TX20Speed
                                MaxGiornaliera</b> km/h</td></tr>
            <tr><td>Mensile</td><td><b>@ViewBag.DatiTX20.TX20SpeedMax
                                Mensile </b> km/h</td></tr>
            <tr><td>Annuale</td><td><b>@ViewBag.DatiTX20.TX20SpeedMax
                                Annuale </b> km/h</td></tr>
        </table>
    </div>
</div>
```

Le parti di codice analizzate fino ad ora si limitano alla semplice visualizzazione di informazioni, prassi che rappresenta il 99% delle esigenze delle richieste da parte dell'utente finale. La forma in cui si rappresentano i dati ha sicuramente un forte impatto visivo, ed il più delle volte è il biglietto di vista dell'intero sito web. La grafica ed il design esulano da questa trattazione, ma rappresentare in modo semplice ed accattivante le letture dei venti può permettere un salto di qualità all'intera piattaforma web.

La parte finale di codice per la rappresentazione dei dati deve inserire le letture in un primo grafico nel quale si riportano tutte le velocità rilevate del vento, ed in un secondo grafico, a rosa dei venti, nel quale si indica la direzione del vento sia come punti cardinali, sia in gradi. Ovviamente deve avvenire un controllo, come fatto in precedenza, sulla presenza di letture giornaliere, quindi è d'obbligo il testing della variabile "dataOdierna". Per permettere la rappresentazione dei dati nei rispettivi grafici, si fa uso di uno script Javascript chiamato "grafici.js" che dovrà venire copiato nel direttorio "Scripts". Questo codice JS richiede la creazione di una variabile formattata secondo lo standard JSON ed inizializzata con i dati da visualizzare. Questo fase deve avvenire prima di chiamare "grafici.js". E' importante ricordare che nella classe TX20 è stata definita la proprietà "public DatiPerGrafico[] TX20ElencoParametriTX20 { get; set; }" che permette di definire un vettore di oggetti "DatiPerGrafico", i quali contengono le informazioni delle singole letture giornaliere, informazioni che verranno riportate nei due grafici. Il punto cardine da capire è la necessità di convertire questo array in una stringa in formato JSON. Questa operazione viene effettuata tramite il metodo "Encode()" della classe "Json", il quale richiede il contenuto restituito dalla proprietà "TX20ElencoParametriTX20", motivo per cui è necessario chiamare in causa Razor tramite la chiamata "@ViewBag.DatiTX20.TX20ElencoParametriTX20". L'intera stringa JSON con i dati non deve venire convertita in tag HTML, ma deve essere a disposizione di Javascript per una corretta manipolazione, motivo per cui prima di memorizzarla nella variabile " jsObjElencoVelocita" va passata all'espressione "@Html.Raw" di Razor che non effettua nessun tipo di conversione verso HTML.

```
@if (dataOdierna != "-")
{
  <br /><br />
  <h3> Dettaglio grafici giornalieri</h3>
       <div style="width:90% ; margin:0 auto ;">
       <div id="graficoVento" style="margin:10px;"></div>
       <div id="windrose" style="margin:10px;"></div>
       <script type="text/javascript">
        var jsObjElencoVelocita =
             @Html.Raw(Json.Encode(@ViewBag.DatiTX20.TX20ElencoParametriTX20));
```

La struttura dati necessaria allo script "grafici.js" deve essere chiamata "json" e deve contenere al suo interno altre strutture dati.
Il vettore "wswd" deve contenere in ogni elemento un oggetto il quale definisce delle coppie chiave-valore che rappresentano la coordinata "x" del grafico delle velocità del vento, la misurazione rilevata "y", la direzione del vento espressa come punto cardinale "cardinalPoint" ed infine la direzione espressa in gradi "windDirection".
Il vettore "wswd_rose" definisce un array i cui elementi sono a sua volta vettori. Gli elementi del singolo vettore rappresentano i dati da visualizzare nella rosa dei venti, ossia la direzione in gradi e le velocità del vento.

```
var json =  {
                "wswd": [ ],
                "wswd_rose": [[ ]]
            }
```

Si inizializza un vettore di appoggio chiamato "elencoRose".

```
var elencoRose = [ ]
```

I dati da inserire negli array "wswd" e "wswd_rose" devono venire caricati dalla variabile "jsObjElencoVelocita", accedendo in modo diretto tramite un indice, visto che non è stata effettuata nessun tipo di conversione verso HTML. Sfruttando l'accesso diretto basta implementare un ciclo iterativo, per un numero di iterazioni pari alla lunghezza della stringa "jsObjElencoVelocita" che essendo in formato JSON è trattabile come un vettore. Viene utilizzata la proprietà "length" per determinare il numero degli elementi vettoriali. L'accesso al singolo blocco di informazione JSON avviene ovviamente tramite l'indice "i".
La singola lettura del blocco JSON comporta la memorizzazione di tutte le informazioni in una struttura dati temporanea chiamata "jsonTmp" che contiene i medesimi campi della struttura "json" necessaria allo script "grafici.js". Segue il codice.

```
for (i=0; i<jsObjElencoVelocita.length; i++)
{
    var jsonTmp =
```

```
{
    "wswd": [
    {
        "x": jsObjElencoVelocita[i].conversioneOraPerStepX,
        "y": jsObjElencoVelocita[i].velocitaVento,
        "cardinalPoint": jsObjElencoVelocita[i].direzioneVento,
        "windDirection": jsObjElencoVelocita[i].direzioneVentoInGradi
    }                    ],
   "wswd_rose": [jsObjElencoVelocita[i].direzioneVentoInGradi,
                                        jsObjElencoVelocita[i].velocitaVento]
};
```

Come accennato in precedenza, nell'array "wswd" si memorizza un oggetto con le proprietà "x", "y", "cardinalPoint" e "windDirection". I rispettivi dati vengono prelevati dal vettore JSON "jsObjElencoVelocita" all'indice "i", accedendo ai tag JSON "conversioneOraPerStepX", "velocitaVento", "direzioneVento" e "direzioneVentoInGradi". I nomi delle proprietà dell'oggetto devono essere "x", "y", "cardinalPoint" e "windDirection" perché lo script "grafici.js" utilizza questi nomi. Stesso discorso per l'array "wswd_rose". Terminata l'inizializzazione della struttura dati "jsonTmp" con i dati del singolo blocco JSON, è necessario provvedere alla memorizzazione del singolo array "wswd" di "jsonTmp" in "wswd" della struttura dati cardine "json". In tale caso essendoci un unico oggetto in "wswd" di "jsonTmp", si accede all'intero contenuto tramite l'indice "0", mentre la scrittura in "wswd" di "json" deve avvenire con indice "i", dovendo quest'ultima contenere tutti gli oggetti delle letture da visualizzare.

```
json.wswd[i]=jsonTmp.wswd[0]
```

Per l'array "wswd_rose", che contiene al suo interno un singolo vettore, si memorizza l'unico array presente nella struttura dati di appoggio "elencoRose", accedendo a quest'ultima con indice "i" così da avere al termine del ciclo una sequenza di array in "elencoRose".

```
elencoRose[i] = jsonTmp.wswd_rose
```

Al termine di ciclo iterativo, il vettore "wswd" di "json" è correttamente inizializzato, mentre resta da memorizzare in "wswd_rose" di "json" tutto il contenuto presente nel vettore di appoggio "elencoRose". Il trasbordo dei dati avviene con una semplice assegnazione. Nel momento in cui la struttura dati complessa "json" è correttamente inizializzata con tutte le informazioni da visualizzare, è necessario richiamare lo script "grafici.js" che crea nella pagina web il grafico di rilevamento della velocità e della rosa dei venti.

```
        }

        json.wswd_rose=elencoRose
        ...
       </script>
       <script src='~/Scripts/grafici.js'></script>
```

Le restanti parti di puro codice HTML non vengono discusse singolarmente perché sono molto intuitive. Segue il listato completo della pagina "index.cs.html" il quale non fa uso di nessun tipo di layout per la pagina web, motivo per cui in Razor si imposta la relativa proprietà a "null" così da eliminare il layout di default che verrebbe ereditato dalla vista.

```
@{
    Layout = null;
}
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
"http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="it" version="XHTML+RDFa 1.0"
dir="ltr"
    xmlns:content="http://purl.org/rss/1.0/modules/content/"
    xmlns:dc="http://purl.org/dc/terms/"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:og="http://ogp.me/ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
```

```
    xmlns:sioc="http://rdfs.org/sioc/ns#"
    xmlns:sioct="http://rdfs.org/sioc/types#"
    xmlns:skos="http://www.w3.org/2004/02/skos/core#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

<head profile="http://www.w3.org/1999/xhtml/vocab">
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta http-equiv="content-language" content="it" />

  <title>Stazione meteo Raspberry Pi 3</title>

  <link href="~/Content/style21b26.css?v2" rel="stylesheet" type="text/css" />
  <link rel="stylesheet" type="text/css" href="~/Content/default1b26.css?v2">
  <link type="text/css" rel="stylesheet" media="all" href="~/Content/jquery.cluetip.css" />
  <link rel="stylesheet" href="~/Content/leaflet.css" />
  <script type="text/javascript" src="~/Scripts/jquery.min.js"></script>
  <script type="text/javascript" src="~/Scripts/highcharts.js"></script>
  <script type="text/javascript" src="~/Scripts/highcharts-more.js"></script>
  <script type="text/javascript" src="~/Scripts/exporting.js"></script>
  <script type="text/javascript" src="~/Scripts/highslide.min.js"></script>
  <script type="text/javascript">

      hs.outlineType = "outer-glow";

      $(document).ready(function() {
        $(".cluetips").cluetip({
            splitTitle: "|",
            showTitle: false,
            clickThrough: true
        });
      });
  </script>
  <script src="~/Scripts/leaflet.js"></script>
```

```
</head>

<body class="two-sidebars">

   <link rel="stylesheet" type="text/css" href="~/Content/menuc4ca.css?1" />
   <div id="navigation-mnw">
      <div class="region region-navigation">
         <div id="block-jquerymenu-1" class="block block-jquerymenu">
            <div class="content"> </div>
         </div>
      </div>
   </div>

   <div id="header" style="min-width: 1090px;">
     <a href="https://www.raspberrypi.org/"><img src="~/Content/Images/logo-header.png"
         alt="RaspberryNetwork" /> </a>
   </div>

   <div class="main" style="min-width: 1090px;">
      <div id="content">
         <div class="section">
            <br />
            <h1>Stazione meteo di Raspberry Pi 3</h1>
            <h2>Rilevazioni velocità del vento di Riva del Garda (TN)</h2>

            <br clear="both" />

            <div style="float:right">
              <div class="g-plusone" data-size="medium"
                    data-href="~/Views/Meteo/index.cshtml"></div>
                <script type="text/javascript">
                     window.___gcfg = {lang: 'it'};
```

```
            (function() {
              var po = document.createElement('script'); po.type = 'text/javascript';
                        po.async = true;
              po.src = '~/Scripts/plusone.js';
              var s = document.getElementsByTagName('script')[0];
              s.parentNode.insertBefore(po, s);
            })();
    </script>
</div>

<br clear="both">

@{var dataOdierna = ViewBag.DatiTX20.Data;}

@if (dataOdierna != "-")
{
    <h3 style="display: inline;">
        Dati in diretta(aggiornati alle @ViewBag.DatiTX20.Tempo del
                        @ViewBag.DatiTX20.Data)
    </h3>
        <span style="display: inline;border: 1px solid #77DD77;background-color:
                    #77DD77;border-radius: 7px;padding: 1px;">
            ONLINE
        </span>
}
else
{
    <h3 style="display: inline;">
        Anemometro al momento nello stato di
    </h3>
        <span style="display: inline;border: 1px solid #77DD77;background-color:
                    #FF0000;border-radius: 7px;padding: 1px;">
            OFFLINE
```

```
        </span>
}

<br /><br />
<div class="datagrid">
   <table>
      <col />
      <col />
      <col />
      <col />
      <col />
      <tbody>
         <tr>
            @if (dataOdierna != "-")
            {
               <td class="headr"><img style="margin: -10px 0 0 0;"
                    src="~/Content/Images/wind.png" alt="Vento" />Vento
               </td>
               <td class="subhead"><span>@ViewBag.DatiTX20.TX20Speed
                     km/h</span><br />
               </td>
               <td class="subhead"><span>@ViewBag.DatiTX20.TipoDiVento
                     </span><br />
               </td>
               <td style="color:#287233">Direzione attuale
                  <strong>@ViewBag.DatiTX20.TX20Direction
                  </strong>
               </td>
               <td style="color:#D53032"><img src="~/Content/Images/up2.png"
                  alt="" /> <strong>@ViewBag.DatiTX20.TX20SpeedMaxGiornaliera
            km/h</strong> <br />alle @ViewBag.DatiTX20.TempoTX20SpeedMax
              </td>
            }
```

```
            </tr>
        </tbody>
    </table>

</div>

@if (dataOdierna != "-")
{
    <br /><br />
    <h3> Dettaglio grafici giornalieri</h3>

    <div style="width:90% ; margin:0 auto ;">
        <div id="graficoVento" style="margin:10px;"></div>
        <div id="windrose" style="margin:10px;"></div>

        <script type="text/javascript">

    var jsObjElencoVelocita =
     @Html.Raw(Json.Encode(@ViewBag.DatiTX20.TX20ElencoParametriTX20));

            var json =
                {
                    "wswd": [ ],
                    "wswd_rose": [[]]
                }

            var elencoRose = []

            for (i=0; i<jsObjElencoVelocita.length; i++)
            {
                var jsonTmp = {
                    "wswd": [
                        {
```

```
                    "x": jsObjElencoVelocita[i].conversioneOraPerStepX,
                    "y": jsObjElencoVelocita[i].velocitaVento,
                    "cardinalPoint": jsObjElencoVelocita[i].direzioneVento,
                    "windDirection": jsObjElencoVelocita[i].direzioneVentoInGradi
                  }
                  ],
                    "wswd_rose": [jsObjElencoVelocita[i].direzioneVentoInGradi,
                                  jsObjElencoVelocita[i].velocitaVento]
        };

        json.wswd[i]=jsonTmp.wswd[0]
        elencoRose[i] = jsonTmp.wswd_rose
    }

    json.wswd_rose=elencoRose


Highcharts.setOptions({
        chart : {
        height: 400
    },
        plotOptions: {
        TurboThreshold: 0 ,
        series: {
        animation: false ,
        connectNulls: true
    },
        spline: {
        pointInterval: 15 * 60 * 1000,
        pointStart: Date.UTC(1970, 00, 01, 10, 00, 00),
        marker:{
        radius: 0,
        states: {
```

```
    hover: {
    lineColor: '#D53032',
    fillColor: 'white',
    lineWidth: 2,
    radius: 3
}
}
},
    states: {
    hover: {
    lineWidth: 2
}
}
}
},
    tooltip : {
    xDateFormat: 'Ore %H:%M',
    crosshairs: true,
    shared: true
},
    credits: {
    enabled: false
},
    yAxis: {
    title: {
    text: null
}
},
    lang: {
    decimalPoint: ',',
    thousandsSep: '.',
    months: [
        'Gennaio', 'Febbraio', 'Marzo', 'Aprile', 'Maggio', 'Giugno',
```

```
                    'Luglio', 'Agosto', 'Settembre', 'Ottobre', 'Novembre', 'Dicembre'
                        ],

                weekdays: ['Domenica', 'Lunedì', 'Martedì', 'Mercoledì', 'Giovedì',
                            'Venerdì', 'Sabato'],
                resetZoom: 'Reimposta lo zoom'
            }
            });
        </script>

        <script src='~/Scripts/grafici.js'></script>
    </div>
}

<script src="~/Scripts/jquery.cluetip.min.js"></script>
</div>
</div>

<div class="region region-sidebar-first column sidebar">
<div class="section">
<div id="block-block-3" class="block block-block even">
<div class="content" style="margin-top:20px;">
<div id="news" class="boxed">
<h3 class="block-title"><img src="~/Content/Images/wind.png">
    Dettagli tecnici
</h3>
<table width="99%" class="toolsidebar" cellspacing="0">
<tr>
<td>Modello</td>
<td>La Cross TX20</td>
</tr>
<tr>
<td>Ubicazione</td>
```

```
                    <td>Campo aperto</td>
                  </tr>
                  <tr>
                    <td>Altitudine</td>
                    <td>65</td>
                  </tr>
                </table>
              </div>
              <br />
              <br />
              <br />
              <div class="box3">
                <div class="left_b">
                  <span class="w"><b>Vento </b> </span><br />
                  <table cellpadding="1" class="toolsidebar">
                    <tr><td>Giornaliero</td><td><b>@ViewBag.DatiTX20.TX20Speed
                          MaxGiornaliera</b> km/h</td></tr>
                    <tr><td>Mensile</td><td><b>@ViewBag.DatiTX20.TX20SpeedMax
                          Mensile </b> km/h</td></tr>
                    <tr><td>Annuale</td><td><b>@ViewBag.DatiTX20.TX20SpeedMa
                          xAnnuale </b> km/h</td></tr>
                  </table>
                </div>
              </div>
              <br />
              <br />
            </div>
          </div>
        </div>
      </div>

      <br clear="all" />
```

```
    <div id="footer">
      <div class="region region-footer">
        <div id="block-block-7" class="block block-block first last odd">
          <div class="content">
            <div style="float: left; font-size: 13px; padding: 20px">
              <p>  <strong>Giuliano Pellegrini Parisi</strong><br />
                <a class="footer_link"
                  href="mailto:s203675@studenti.polito.it">
                          s203675@studenti.polito.it</a>  </p>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Prima di procedere alla pubblicazione della WebAppTX20 su Azure, è necessario copiare le immagini, i fogli di stile CSS e i Javascript utilizzati nella pagina.

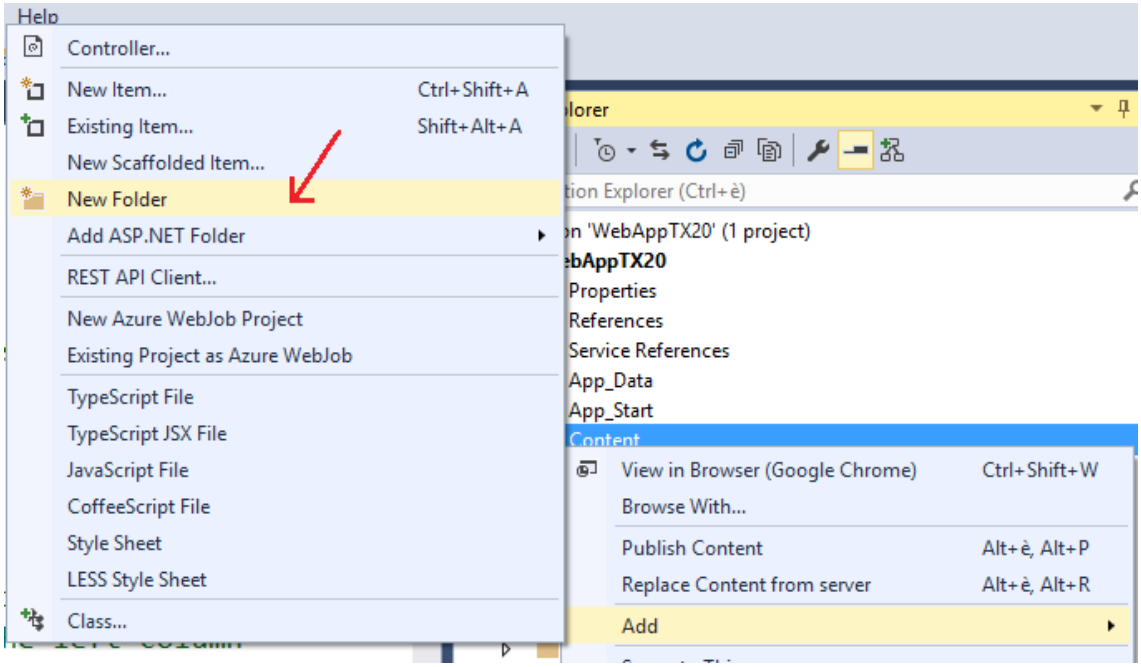


Fig.379

In Fig.379 si procede a creare una nuova directory all'interno del direttorio di default "Content". Questa nuova cartella viene chiamata "Images", come da Fig.380.

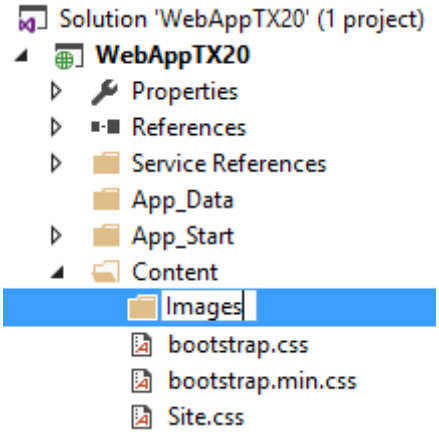


Fig.380

Copiare le immagini del logo della pagina e delle icone, dalla cartella locale "Immagini" nel direttorio appena creato. L'operazione è banale, ma per completezza viene riportata in Fig.381. Il risultato della copia è visibile in Fig.382.

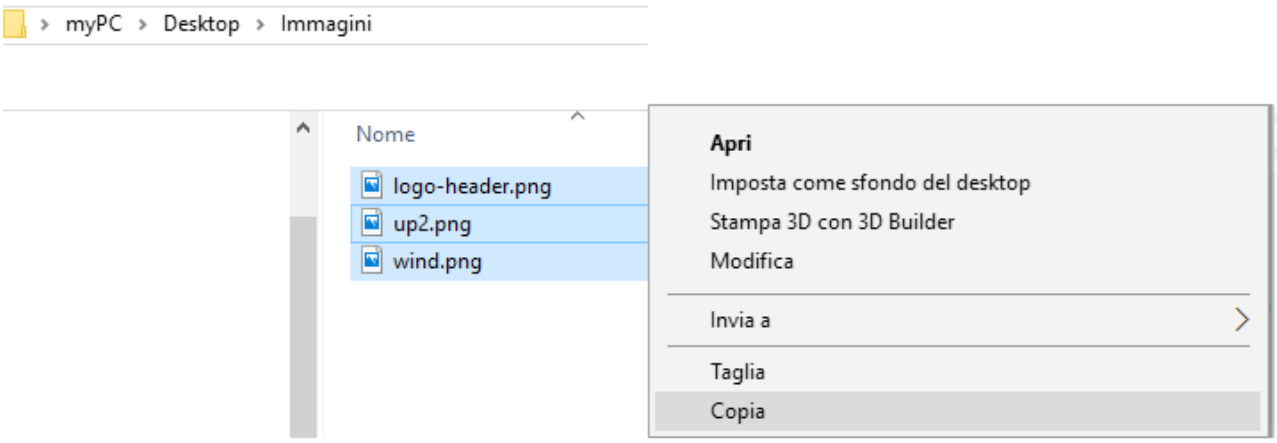


Fig.381

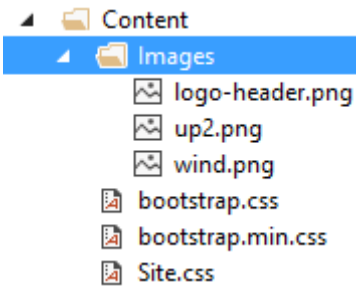


Fig.382

Successivamente si procede alla copia dei fogli di stile CSS dal direttorio locale come da Fig.383.

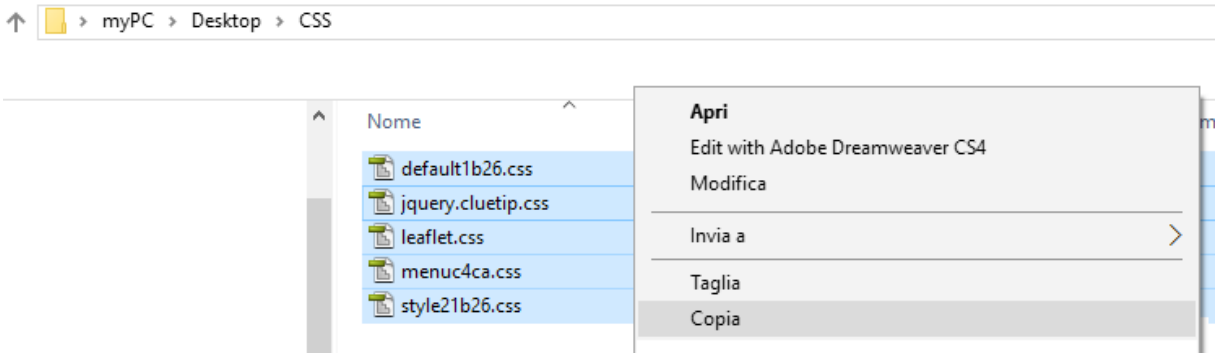


Fig.383

La destinazione dei files è all'interno della directory "Content" come da Fig.384.

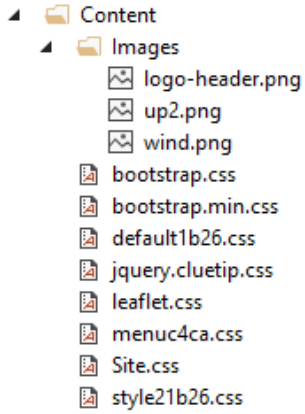


Fig.384

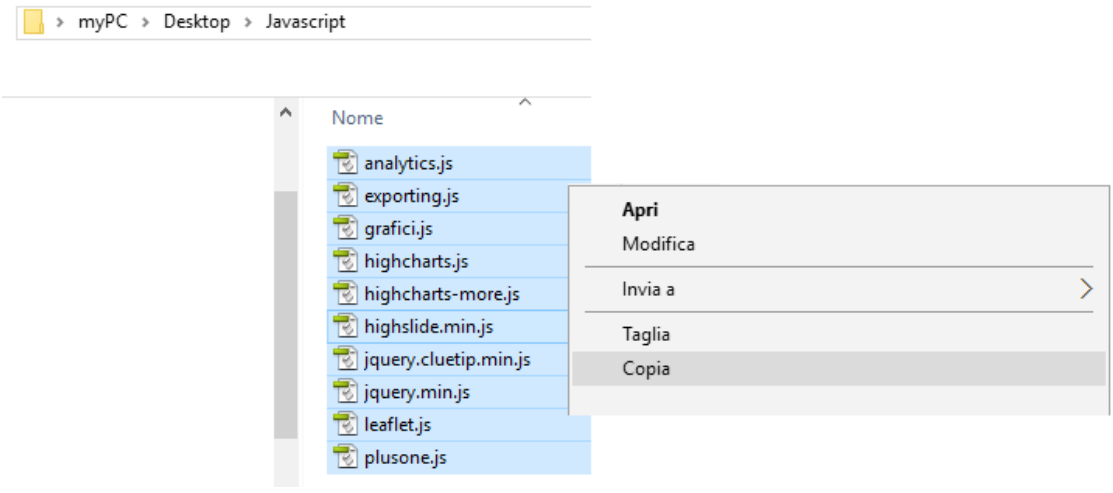


Fig.385

Gli ultimi files da copiare sono l'elenco locale dei Javascript come da Fig.385. La destinazione è la directory "Scripts" come da Fig.386.

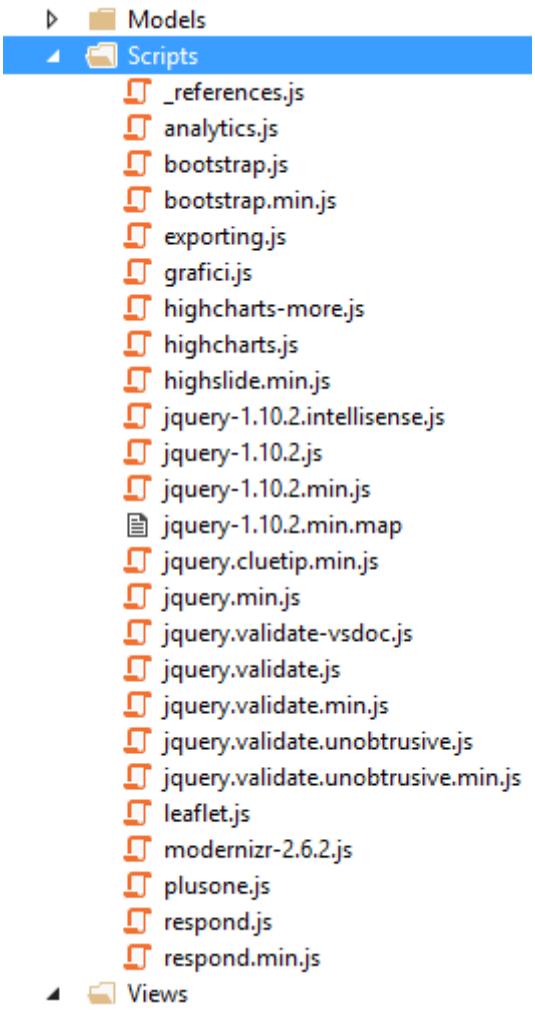


Fig.386

L'intero progetto è pronto per la pubblicazione, quindi non resta che selezionare la WebAppTX20, come in Fig.387, per poi col tasto destro del mouse selezionare la voce "Publish..." come in Fig.388.

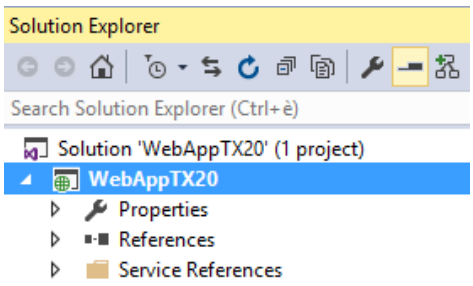


Fig.387

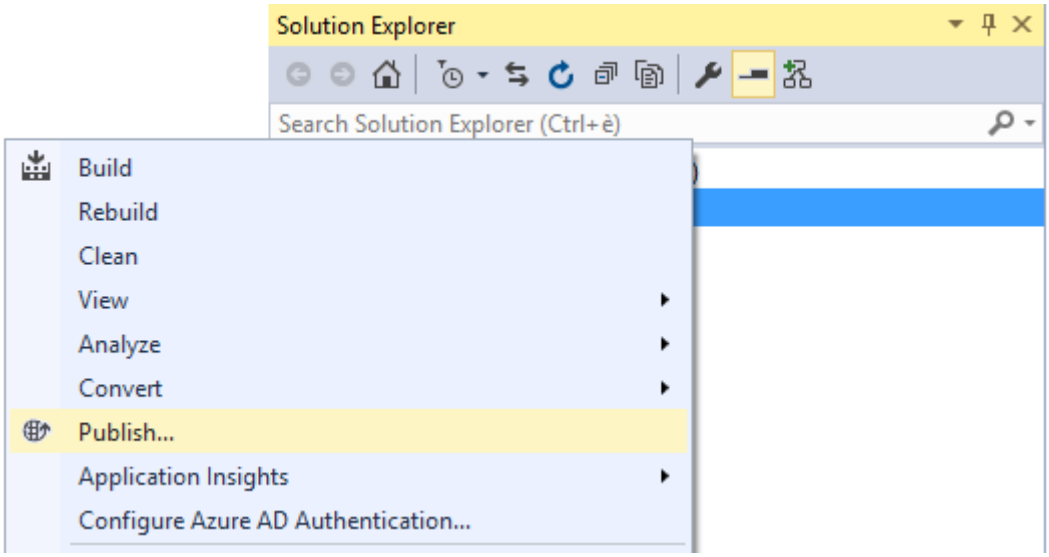


Fig.388

Il processo di pubblicazione impiega del tempo anche in relazione al numero di files da trasferire verso Azure. In Fig.389 alcune fasi della copia remota.

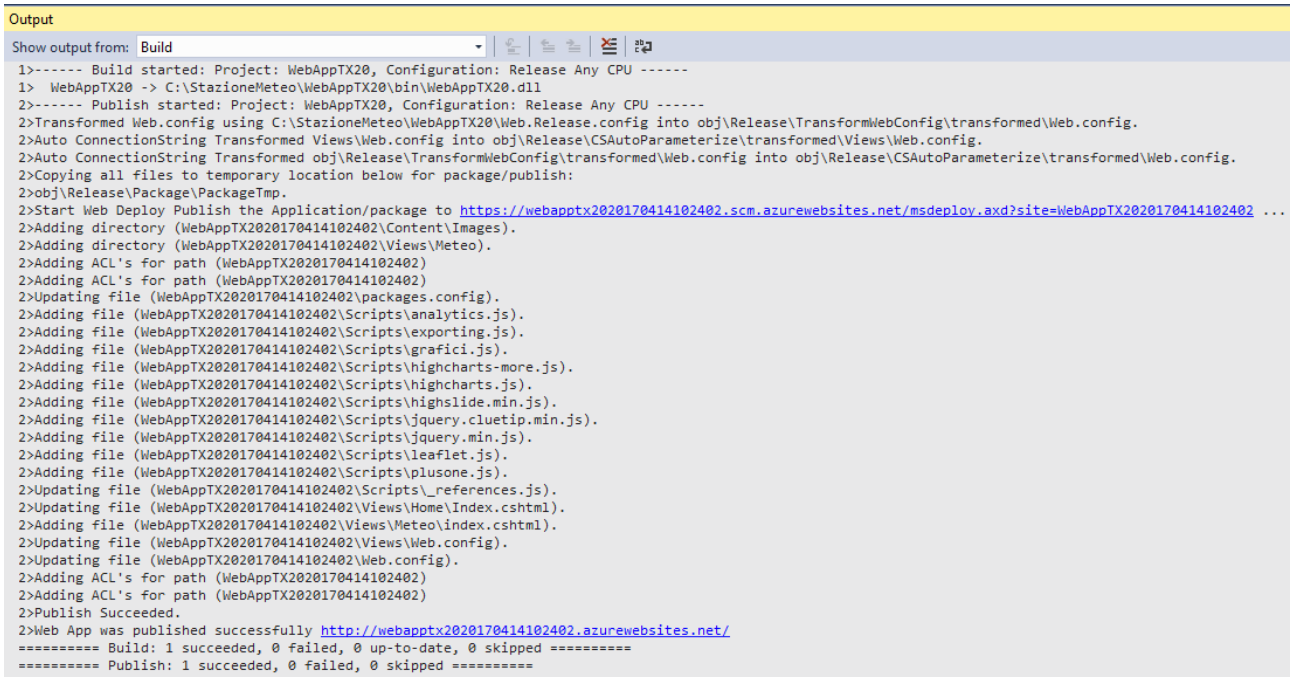


Fig.389

E' interessante notare che al termine della copia, il meccanismo di pubblicazione indica la URL ufficiale da utilizzare per l'impiego dalla Web App che risulta essere http://webapptx2020170414102402.azurewebsites.net

Prima di verificare l'esecuzione della Web App, conviene rivedere in modo sintetico i files di progetto creati fino a questo momento, ad esclusione dei files immagini, dei fogli di stile e dei Javascript. In Fig.390 si evidenziano i files del progetto MVC, quindi la logica di modello "Business Access Layer" con la classe "TX20BusinessAccessLayer.cs" e la classe contenitore dei dati "TX20.cs", sempre a livello di modello. La classe "DatiPerGrafico.cs" per la memorizzazione dei dati al fine della visualizzazione delle letture nella pagina web e la classe "Azure.cs" per la connessione allo storage BLOB dal quale prelevare i dati in formato JSON. Si evidenzia il controllore con la classe "MeteoController.cs" e la vista con la pagina web "index.cshtml". E' importante ricordare la presenza del file di configurazione "Web.config" che contiene il tag di connessione allo storage.

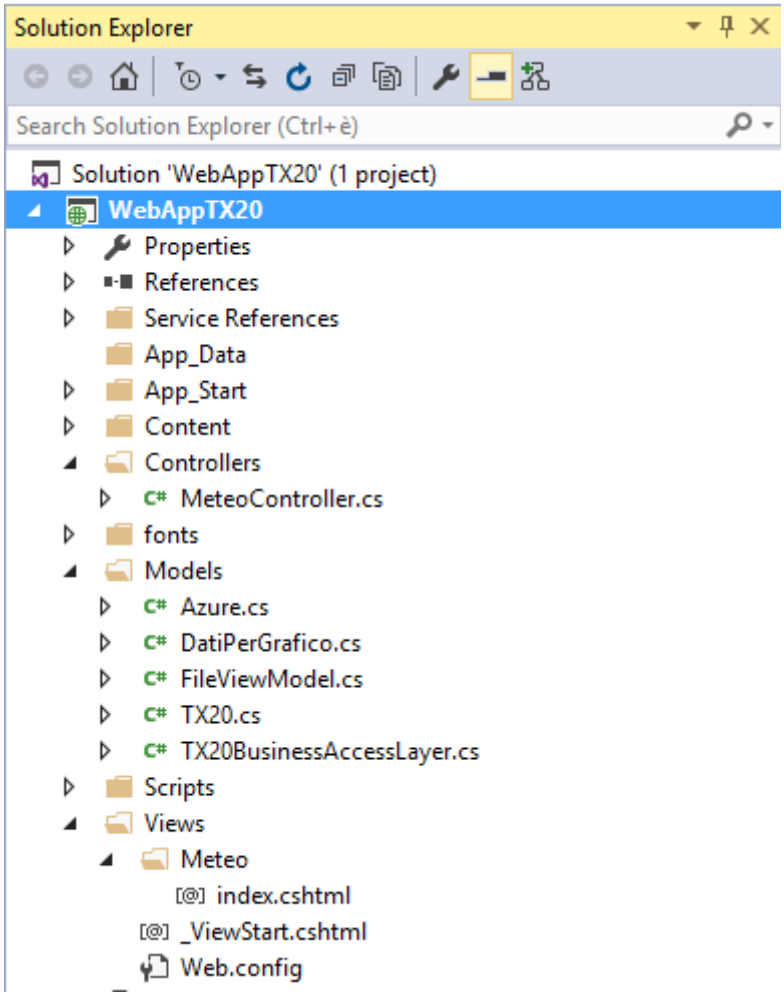


Fig.390

5.4.3 WebApp in esecuzione

Una volta resa pubblica la Web App è sufficiente utilizzare la URL ufficiale per accedere remotamente via browser, come visto nella sezione di costruzione della pagina web. La URL indicata in fase di pubblicazione si riferisce al metodo di default e non a quello personalizzato nel controllore "TX20RivaDelGarda()", motivo per cui l'accesso corretto alla pagina, previo accesso ai dati tramite modello, avviene solamente inserendo la giusta chiamata nella URL, come da Fig.391, in modo che il link risulti essere http://webapptx2020170414102402.azurewebsites.net/Meteo/TX20RivaDelGarda.

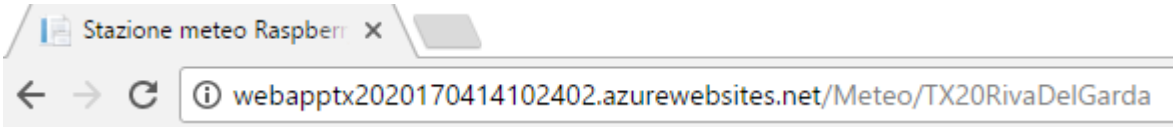


Fig.391

La Fig.392 riporta parte della pagina web con la visualizzazione dei dati di giornata, oltre che la massima velocità mensile ed annuale.

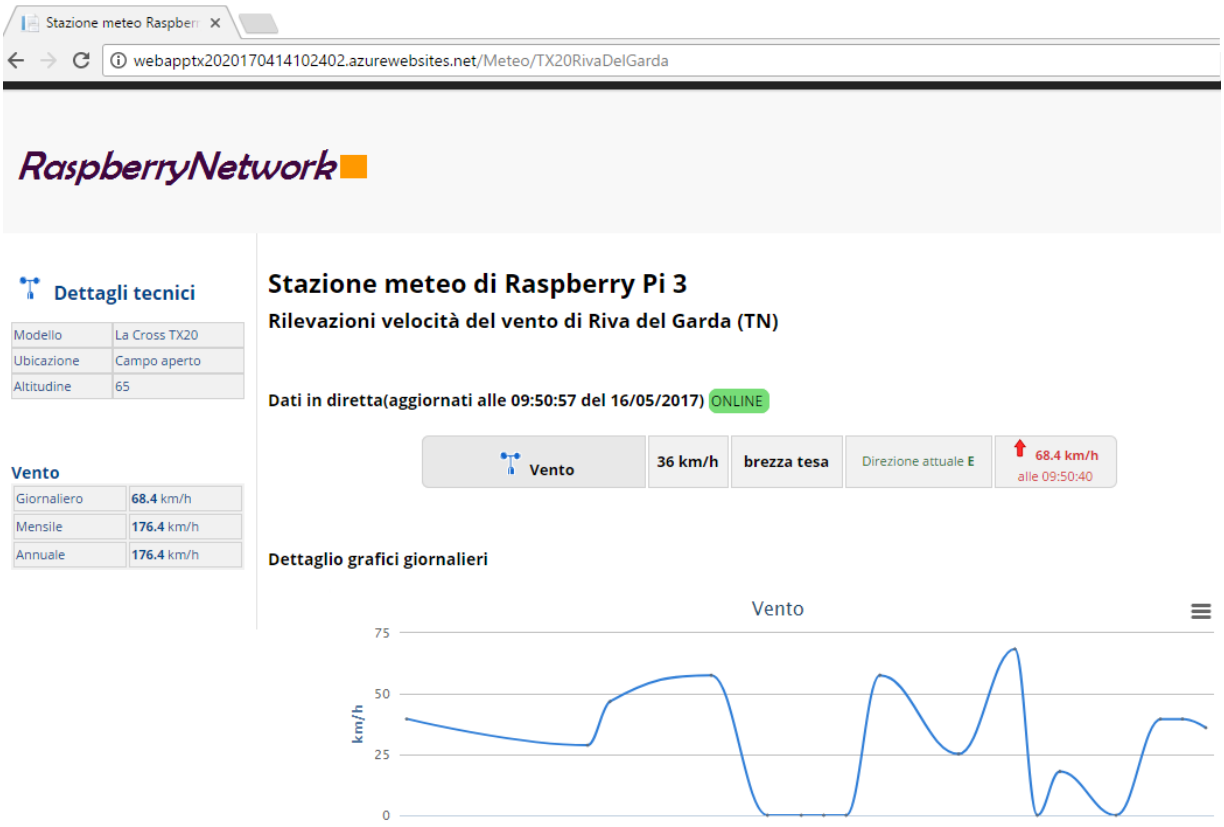


Fig.392

Il grafico possiede un pulsante con il quale è possibile disabilitare/riabilitare la visualizzazione dell'andamento delle velocità, come si vede in Fig.393.

Fig.393

Posizionandosi sopra la singola lettura viene riportata in dettaglio la velocità e l'ora di rilevamento, assieme alla direzione espressa in punti cardinali e gradi. Si veda la Fig.394.

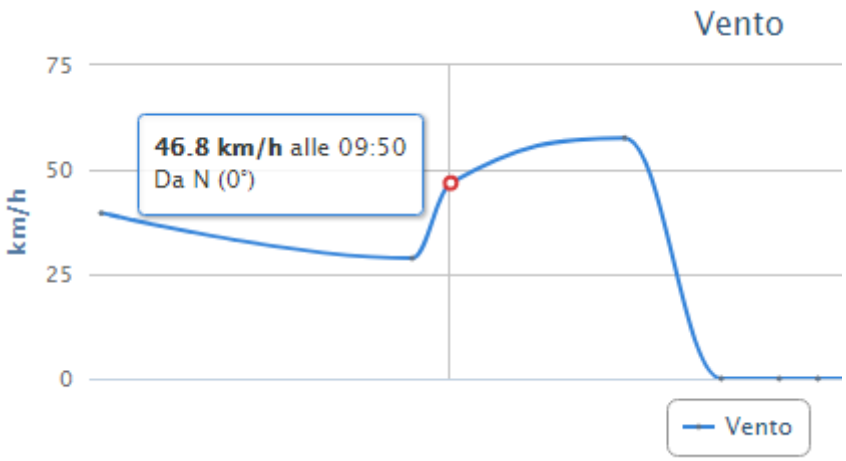


Fig.394

Sul fianco destro del grafico è presente un pulsante che permette di esportare il grafico in alcuni formati, tra cui PNG, JPEG, PDF e formato vettoriale SVG. Si veda la Fig.395. Il grafico a rosa dei venti, presente in Fig.396, visualizza la direzione del vento con la rispettiva velocità. Osservando l'immagine si vede che è presente un pulsante che permette di effettuare l'auto adattamento in base alla velocità, come si vede in Fig.397.

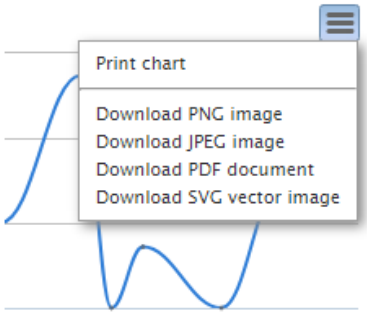


Fig.395

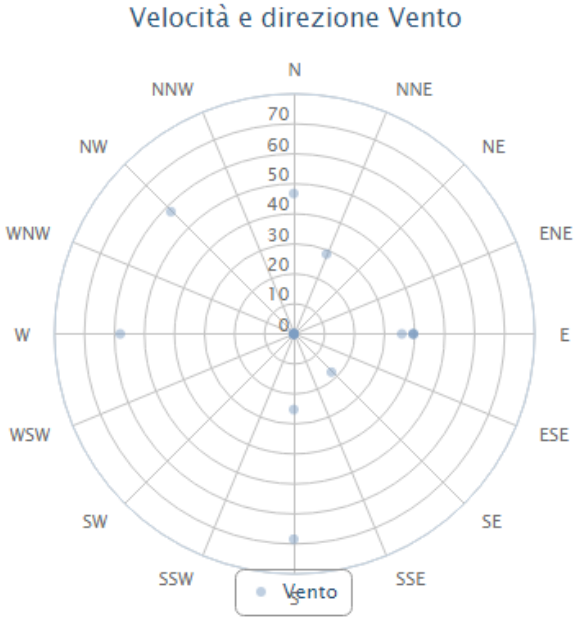


Fig.396

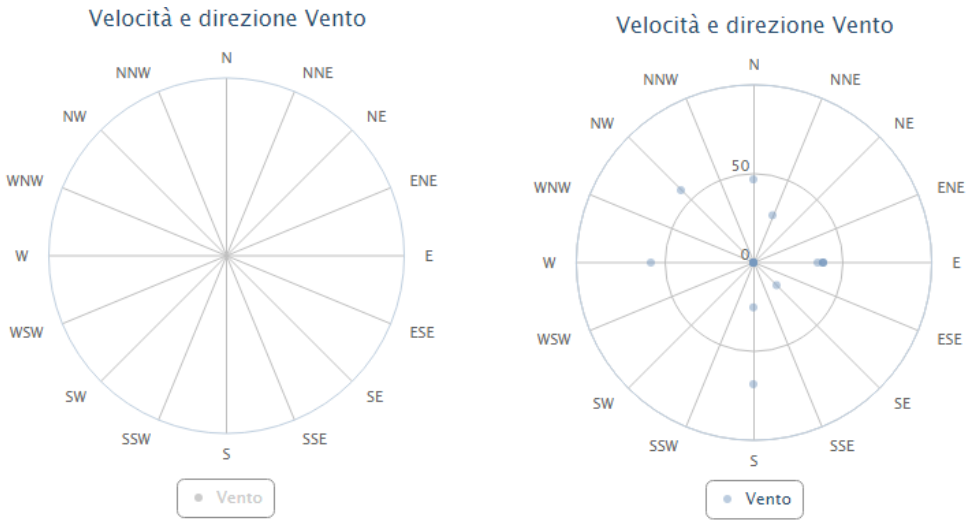


Fig.397

Conclusioni

Le possibilità di utilizzo di schede embedded come il Raspberry Pi hanno come limite la fantasia del maker e la voglia di cimentarsi sulle più disparate realizzazioni. Implementare reti di sensori interconnesse al Pi oggigiorno non comporta grossi problemi, se non l'adattamento in tensione a +3.3V della GPIO, quindi problemi squisitamente elettrotecnici. La possibilità di installare un sistema operativo Windows 10 IoT, permette di creare applicazioni universali UWP e, grazie alla suite Visual Studio 2015 o successiva, in qualsiasi momento è possibile effettuare, se necessario, il debug remoto direttamente dalla postazione di lavoro. Fortunatamente la scrittura di UWP può essere fatta anche con linguaggi gestiti di alto livello, C# ne è una dimostrazione chiara, quindi l'unione di un framewrok .NET Core al paradigma di programmazione ad oggetti offre una logica di sviluppo di primo ordine. Una rete di sensori deve offrire all'utente finale la visualizzazione dei dati, motivo per cui l'interazione device IoT e Cloud rappresenta il nuovo traguardo tecnologico, che permette di avvicinare l'uomo al sensore. L'utilizzo di un cloud come Microsoft Azure offre sicuramente una piattaforma di lavoro seria, robusta ed altamente professionale, così come la possibiltà di interagire con i dati nel cloud tramite Web App o Mobile Web App. Le possibilità realizzative sono molteplici, è compito del programmatore scegliere la soluzione più consona alla richiesta del cliente.

Indice delle tabelle

Indice delle figure

Bibliografia

Donald Norris, *The Internet of Things, McGraw-Hill Education, 2015*
Arshdeep Bagha, *Internet of Things: A Hands-On Approach, VPT 2014*
Adrian Mcewen e Hakim Cassimally, *Designing Internet of Things, John Wiley & Sons Inc 2013*
Mark J. Price, *C# 7 and .NET Core: Modern Cross-Platform Development - Second Edition, Packet Publishing 2017*
Adam Freeman, *Pro ASP.NET Core MVC, Apress 2016*
Enrico Ambrosini, *l'Elettronica - Manuale dei data sheet, Tramontana 2008*
Gaetano Conte ed Emanuele Impollenti, *Elettronica ed Elettrotecnica, Hoepli 2015*