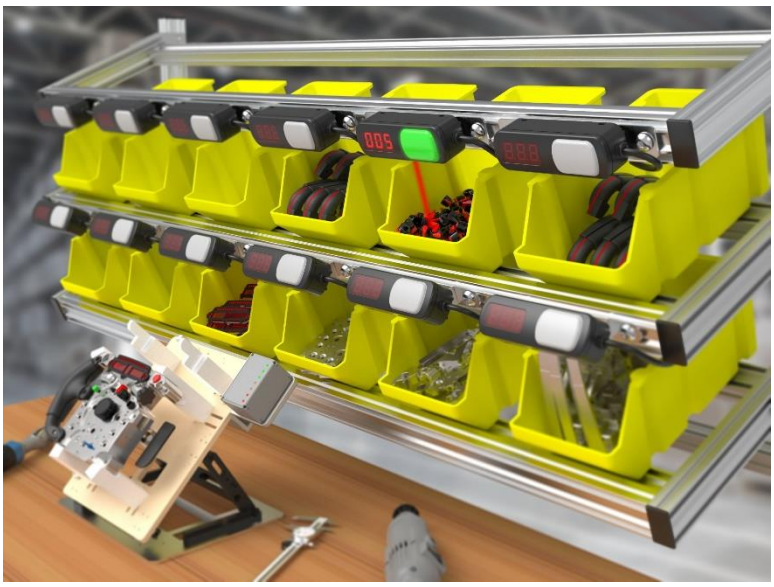
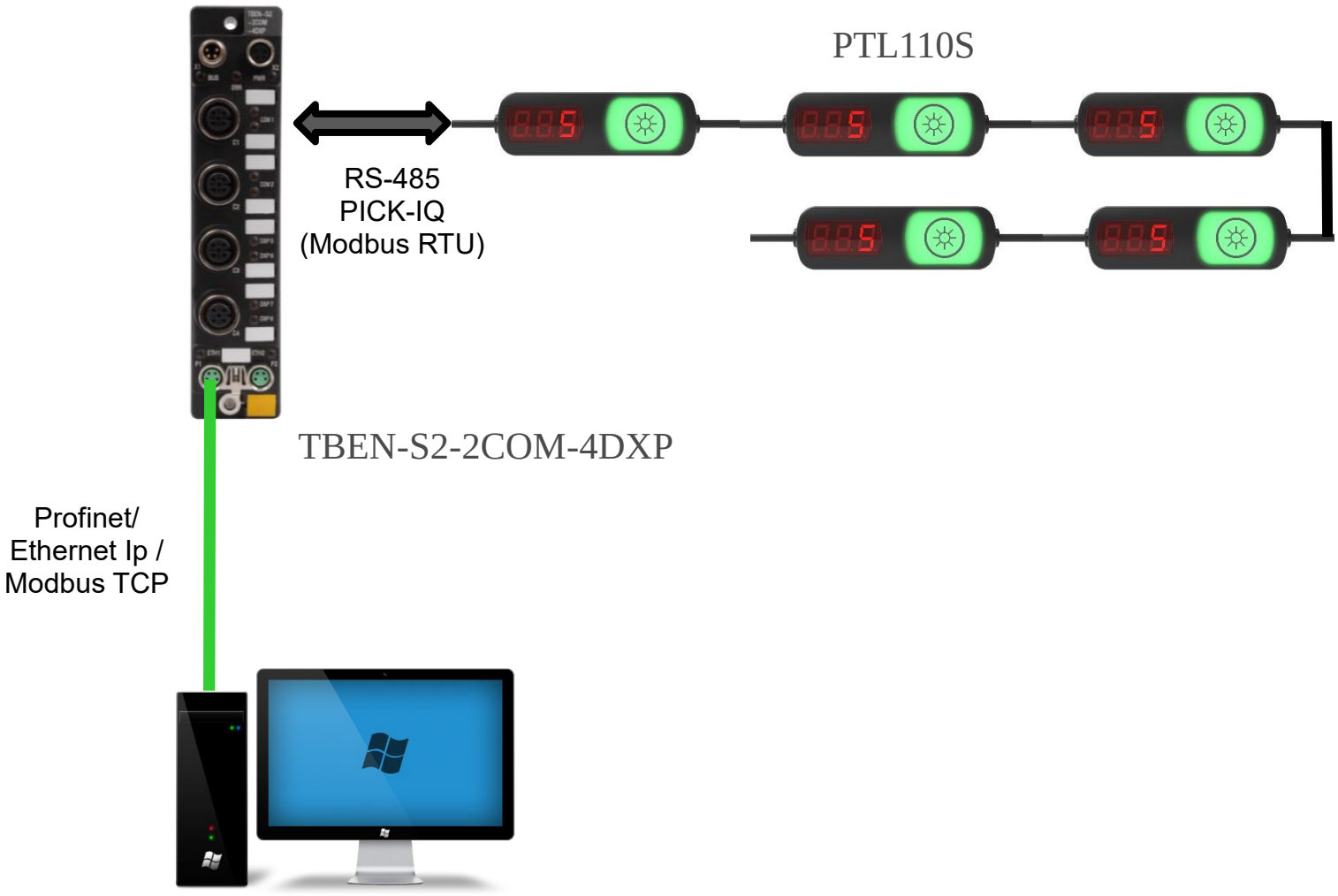
HOW TO

Integration of PTL110S with TBEN-S2-2COM in PC-Application



Sommario

1. Schema generale dell'integrazione:



- TBEN-S2-2COM-4DXP :
  - Fa da gateway, tra il bus di comunicazione a base Ethernet e il protocollo di comunicazione del PTL110S, ovvero il Modbus RTU
  - Ha 2 porte COM, quindi potrebbe gestire 2 linee separate e parallele
  - Ha 4 DXP : segnali digitali universali da poter usare come Ingresso digitale o Uscita (Max 0.5 A)
- PTL110S :
  - Dispostivo di Pick-to-Light evoluto, con comunicazione a base Modbus-RTU, ma ottimizzato e denominato PICK-IQ per le strategie di gestione che migliora i limiti classici del Modbus RTU Base

2. TBEN-S2-2COM-4DXP:

2.1. Datasheet :

2.1.1. http://pdb2.turck.de/repo/media/_en/Anlagen/Datei_EDB/edb_6814031_gbr_en.pdf

2.2. Manuale

2.2.1. http://pdb2.turck.de/repo/media/_en/Anlagen/d301439.pdf

2.3. Dispositivo Multi-protocollo quindi può essere integrato in reti Profinet/Ethernet-Ip, oppure gestito in Modbus TCP sia da controllori dedicati o da PC attraverso .dll o librerie free che si trovano su internet per la gestione del modbus TCP.

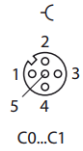
2.3.1. In pratica i comandi Modbus TCP da utilizzare sono:

- WriteMultipleRegister (serve per scrivere i dati nei registri)
- ReadHoldingRegister (serve per recuperare i valori dei registri)
- Questo dovrebbe valere per tutte le librerie, in pratica quando viene lanciato il metodo/Def/Funzione viene eseguita e esce dalla stessa quando è stata eseguita e conclusa e di solito la funzione stessa ritorna i dati in casi della lettura tramite ReadHoldingRegister

2.4. Alimentazione separata per Porte-COM e porte DXP :

2.4.1. Connessione dei dispositivi sulle porte COM segue lo standard quindi basta connettere il connettore delle PTL110s direttamente alla porta COM o tramite una prolunga dritta da 4 fili

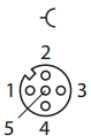
RS485 connection



C0...C1

1 = $V_{aux}1$
2 = TX/RX+
3 = GND V1
4 = TX/RX–
5 = FE

2.4.2. Le porte DXP sono M12 standard con il seguente schema per la gestione :



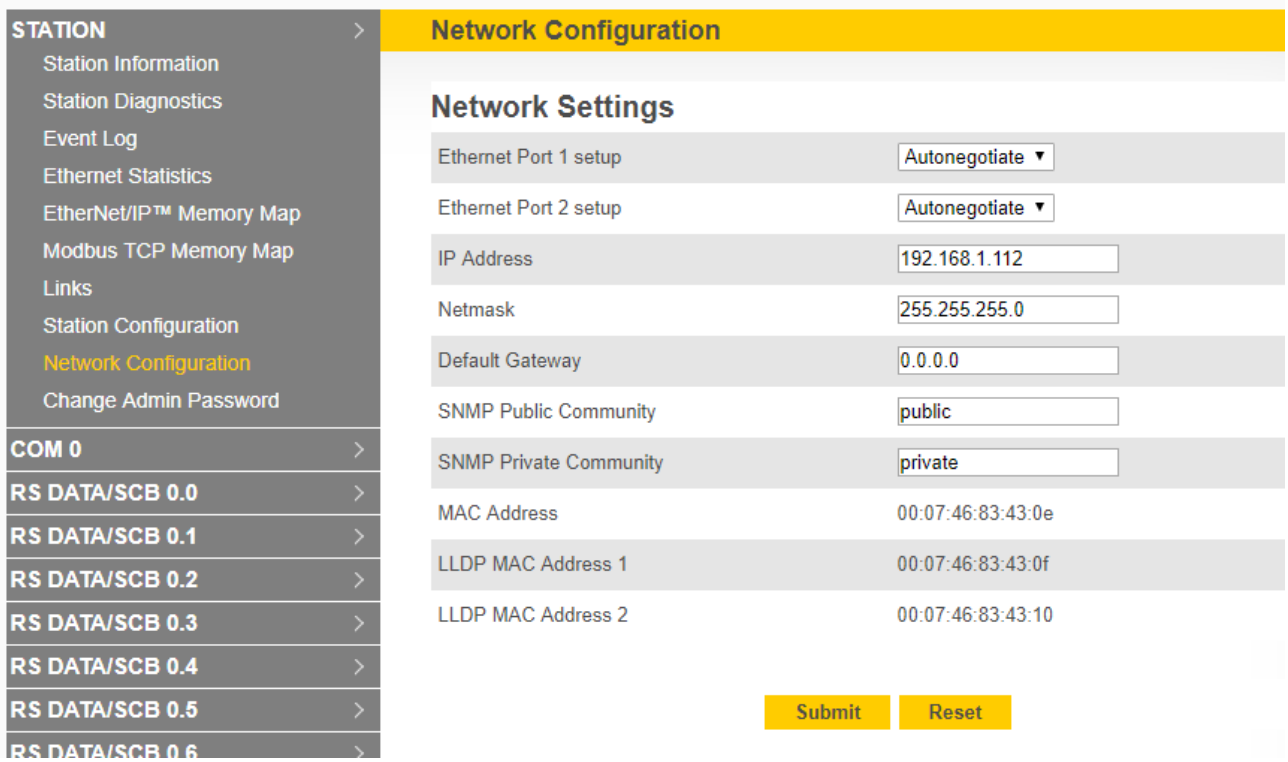
C2...C3

1 = $V_{aux}2$
2 = DXP
3 = GND V2
4 = DXP
5 = FE

2.5. Gestione e uso:

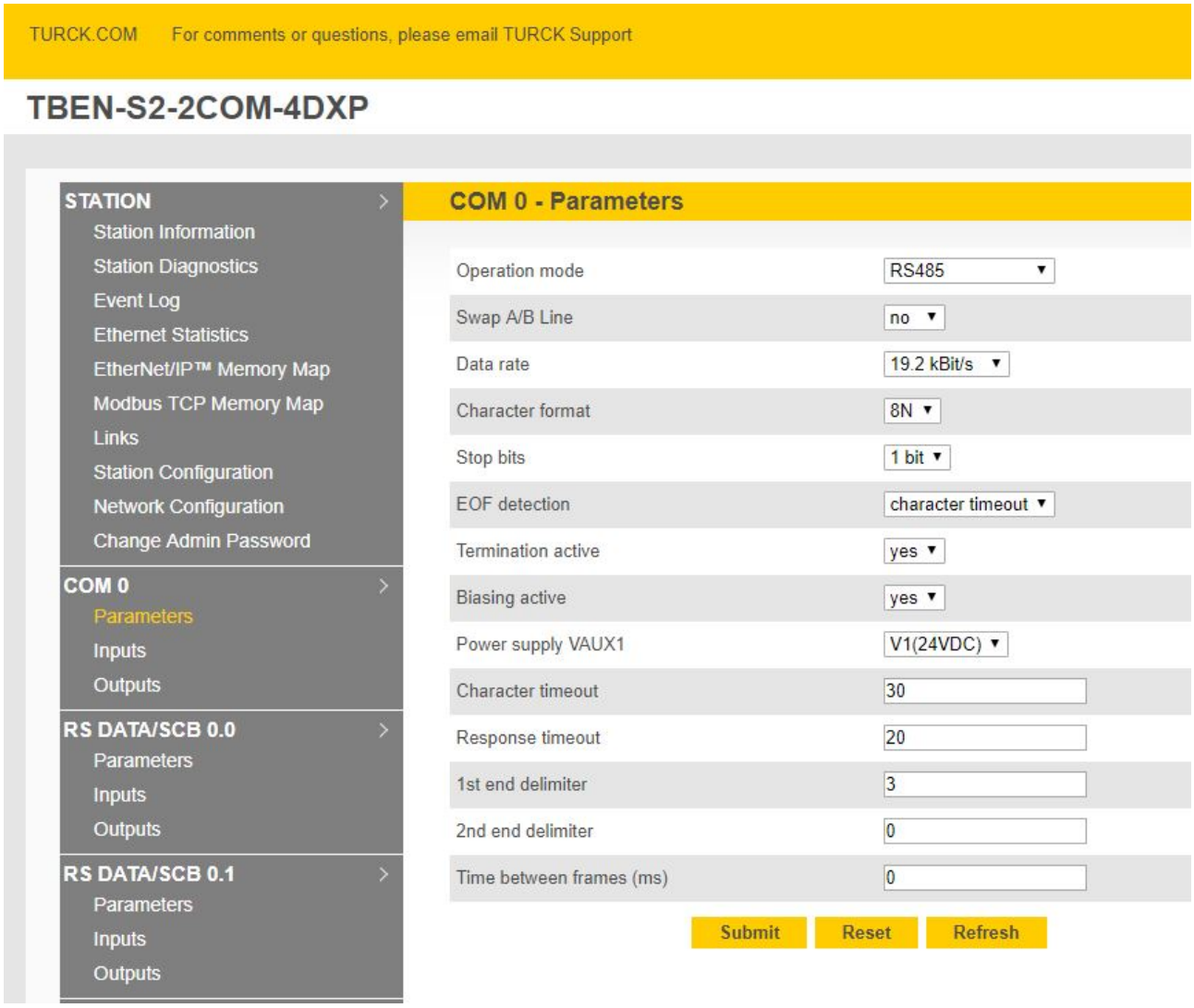
2.5.1. General Setup:

- Il TBEN nasce con indirizzo di default 192.168.1.254, quindi accessibile via browser per gestione-test dello stesso.
- Quindi è possibile impostare i vari parametri necessari al funzionamento e anche testare le varie funzioni e fare diagnostica



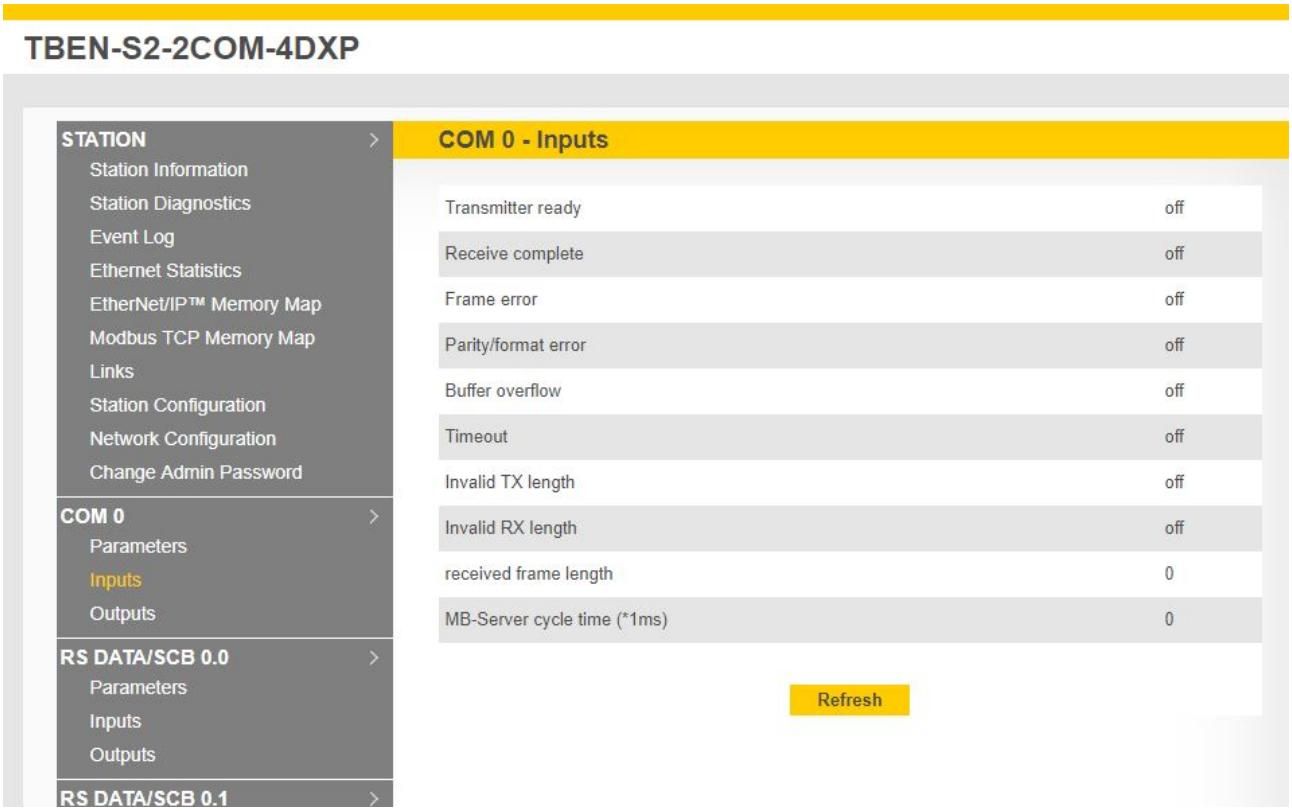
- Dalla pagina web è possibile cambiare l'indirizzo IP
- Fare diagnostica generale del modulo dalla tab Station DIagnostics & Event-Log
- Nella Tab Modbus TCP memory si trova un riassunto dei registri modbus
- Il suggerimento è quello di lasciare tutto di default ad eccezione della pagina relativa alle connessioni di rete da personalizzare in base alla propria rete

2.5.2. COM setup:



- Tutte queste impostazioni potrebbero essere cambiati e modificati anche direttamente da Modbus TCP (a partire dal Registro 16#B000)
- Le impostazioni qua sopra sono quelle necessarie per il funzionamento ottimale con i PTL110S
  - RS485, 19.2 KBit/S, 8N, 1Bit di stop
  - Power Supply VAUX1 = 24VDc, in pratica la porta fornisce alimentazione alla rete PTL110s
  - Response timeout = 20 ms, è importante che sia basso per le logiche del sistema PICK-IQ

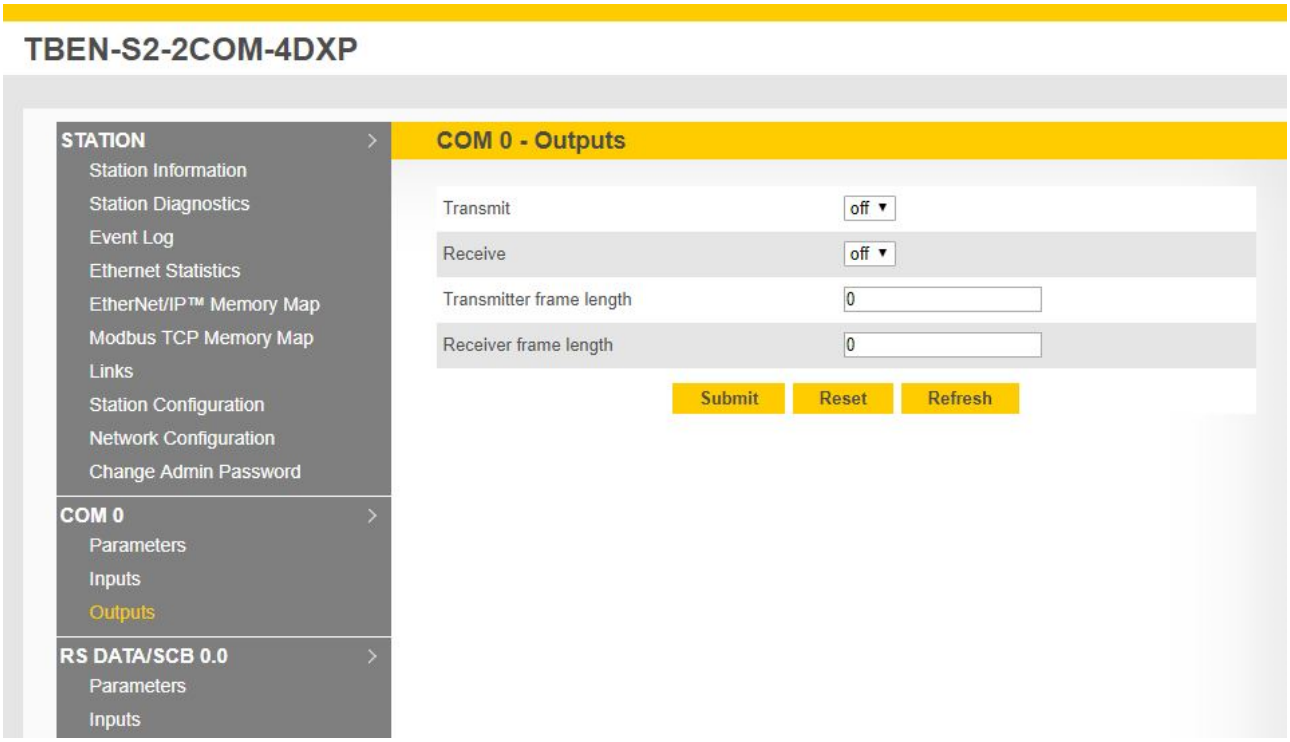
2.5.3. COM Input Status:



- In pratica sono i Bit di Status per la gestione della comunicazione seriale, che dovranno essere interpretati come risposta ai comandi Inviati:
  - Transmitter Ready = TxDone (tramissione completata)
  - Receive Complete = RxDone (Ricezione dati completata)
  - Timeout = Timeout (Tramissione-Ricezione non avenuta)
  - Rceived frame Lenght = Lunghezza in byte di dati ricevuti
  - Errori da tener in considerazione per una diagnostica approfondita ma non indispensabile:
    - Frame Error
    - Parity/format Error
    - Buffer Overflow
    - Invalid TX/RX lenght
- Accesso a questi dati in Modbus TCP:
  - Per COM0 partendo da 16#0000 (COM1 = 16#0063)

Register	Bit pos.	Length	Slot	Module	Parameter
0000	0	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Transmitter ready
0000	1	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Receive complete
0000	2	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Frame error
0000	3	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Parity/format error
0000	4	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Buffer overflow
0000	5	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Timeout
0000	6	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Invalid TX length
0000	7	1	1	Intern-S2-2COM-4DXP	COM 0/Input values/Invalid RX length
0001	0	8	1	Intern-S2-2COM-4DXP	COM 0/Input values/received frame length
0002	0	16	1	Intern-S2-2COM-4DXP	COM 0/Input values/MB-Server cycle time (*1ms)

2.5.4. COM Output Command :



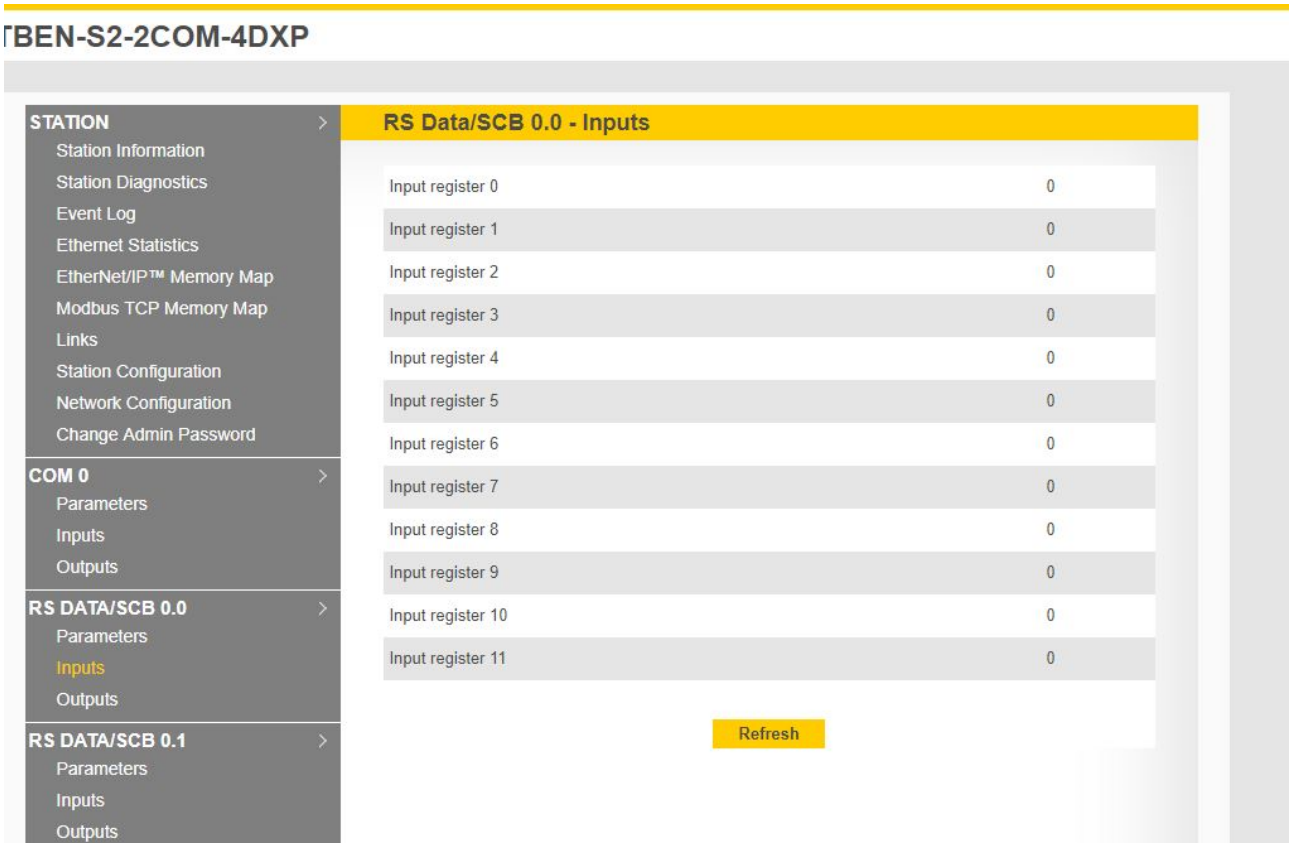
- In pratica sono i bit di comando da gestire per triggerare gli eventi :
  - Trasmit = CmdTx (esegui comando di scritura)
  - Receive = CmdRx (esegui comando di lettura)
  - Transmitter Frame Lenght = LenghtTx (lunghezza in byte Tx)
  - Receive Frame Length = LengthRx (lunghezza in byte attesa in Rx)
- Accesso a questi dati in Modbus TCP:
  - Per COM0 partendo da 16#0800 (COM1 = 16#0863)

Register	Bit pos.	Length	Slot	Module	Parameter
0800	0	1	1	Intern-S2-2COM-4DXP	COM 0/Output values/Transmit
0800	1	1	1	Intern-S2-2COM-4DXP	COM 0/Output values/Receive
0800	6	1	1	Intern-S2-2COM-4DXP	COM 0/Output values/Flush RX buffer *)
0800	7	1	1	Intern-S2-2COM-4DXP	COM 0/Output values/Flush TX buffer *)
0801	0	8	1	Intern-S2-2COM-4DXP	COM 0/Output values/Transmitter frame length
0802	0	8	1	Intern-S2-2COM-4DXP	COM 0/Output values/Receiver frame length

2.5.5. COM data Param :

- Per questa applicazione non deve essere usata

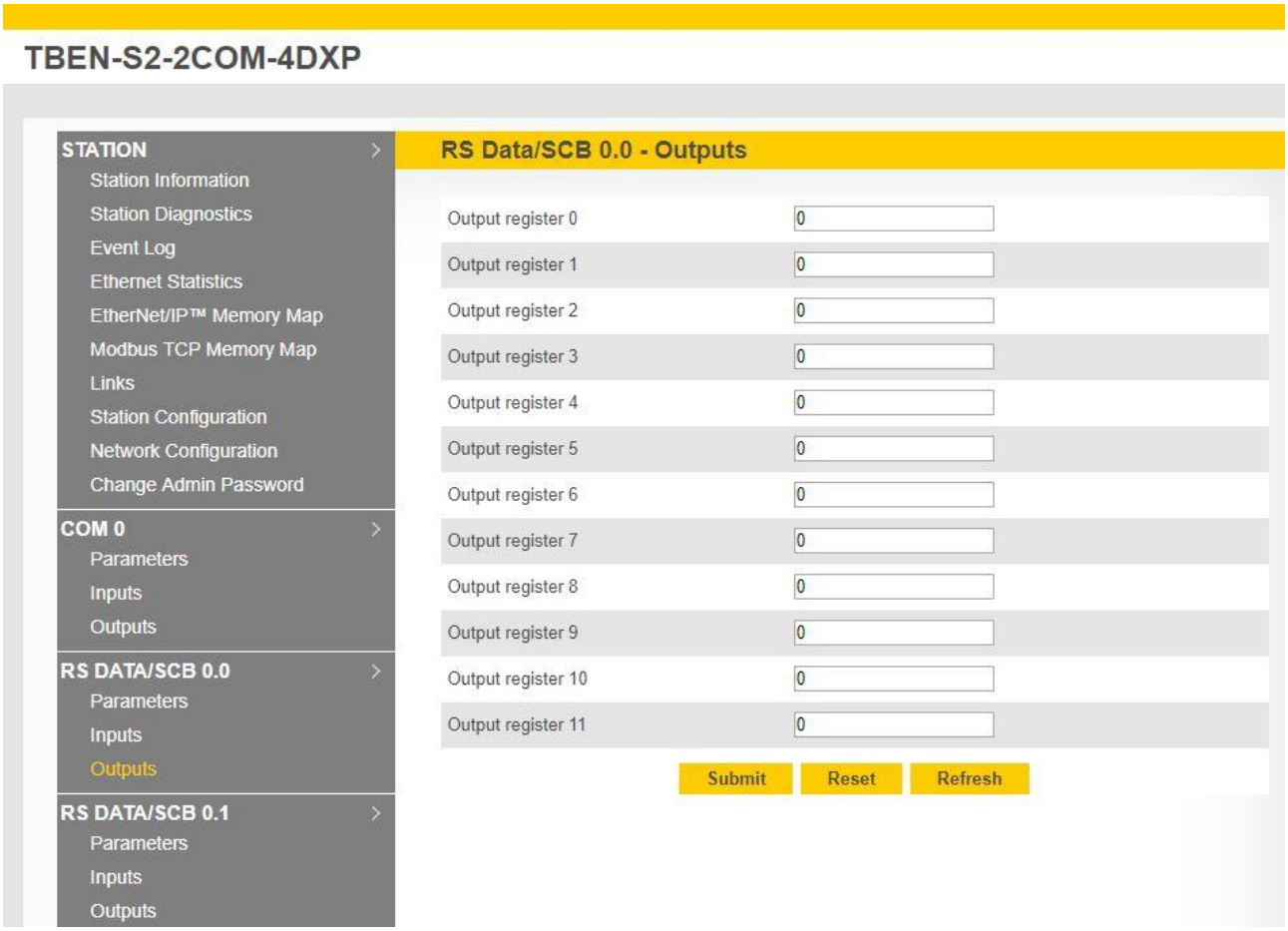
2.5.6. COM data Input :



- In pratica sono 12 Word, ovvero 24 Byte di dati che vengono popolati quando una richiesta di lettura dati verso gli slave viene completata e i dati in risposta vengono popolati qua divise in word.
- Quindi bisogna tener conto di questi limiti quando si fanno delle richieste di lettura (comprensivo dei byte di header e CRC, vedremo dopo nel dettaglio)
- Le word andranno splittate in byte e analizzate di conseguenza in base allo standard Modbus RTU
- Accesso a questi dati in Modbus TCP:
  - Per COM0 partendo da 16#0003 (COM1 = 16#0066) quindi una lunghezza di 12 word

0003	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 0 **)
0003	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 0
0004	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 1 **)
0004	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 1 **)
0005	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 2 **)
0005	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 2 **)
0006	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 3 **)
0006	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 3 **)
0007	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 4 **)
0007	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 4 **)
0008	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 5 **)
0008	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 5 **)
0009	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 6 **)
0009	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 6 **)
000A	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 7 **)
000A	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 7 **)
000B	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 8 **)
000B	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 8 **)
000C	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 9 **)
000C	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 9 **)
000D	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 10 **)
000D	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 10 **)
000E	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 11 **)
000E	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Input register 11 **)

2.5.7. COM data Output :



- In pratica sono 12 Word, ovvero 24 Byte di dati che devono essere popolati con i dati da inviare, prima di gestire i bit di comando che abbiamo visto prima (CmdTX e CmdRx)
- Quindi bisogna tener conto di questi limiti quando si fanno delle richieste di lettura (comprensivo dei byte di header e CRC, vedremo dopo nel dettaglio)
- Le word andranno popolate partendo dai byte in base standard Modbus RTU, e anche twistati i byte delle word per gli standard Little/Big Endian.
- Accesso a questi dati in Modbus TCP:
  - Per COM0 partendo da 16#0803 (COM1 = 16#0866) quindi una lunghezza di 12 word

0803	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 0 **)
0803	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 0
0804	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 1 **)
0804	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 1 **)
0805	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 2 **)
0805	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 2 **)
0806	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 3 **)
0806	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 3 **)
0807	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 4 **)
0807	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 4 **)
0808	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 5 **)
0808	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 5 **)
0809	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 6 **)
0809	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 6 **)
080A	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 7 **)
080A	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 7 **)
080B	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 8 **)
080B	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 8 **)
080C	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 9 **)
080C	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 9 **)
080D	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 10 **)
080D	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 10 **)
080E	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 11 **)
080E	0	16	1	Intern-S2-2COM-4DXP	COM 0 SCBs/SCB 0/Output register 11 **)

2.6. Gestione Ciclica - semplificata:

2.6.1. Per poter gestire il TBEN tenendo conto delle cicliche del Pick-IQ bisogna simulare un simil-ciclo PLC lato programma PC ex:

- Controllo lo stato dei Bit di stato in ingresso (TxDone, RxDone, Timeout,etc)
- Eseguo le mie logiche
- Nel caso di comandi da inviare:
  - Preparo i dati nel Buffer Output
  - Alzo i comandi di CmdTx e/o CmdRx
- Attendo che i comandi verso gli slave vengano eseguiti , ovvero Bit di cui sopra
- In caso negativo, attendo ma vado avanti e ri-inizio il ciclo.
- In caso Positivo :
  - Se il comando era solo un CmdTx , ovvero invio dei comandi e non mi aspetto delle risposte allora finisco
  - Se il comando era CmdTx+CmdRx, allora se il Bit di RxDone è alto vado a valutare i dati presenti nel Buffer Input e li elaboro per gestirli nel programma
  - Quindi abbasso i bit di comando CmdTx e CmdRx
- Quindi posso proseguire con altre logiche o comandi

2.6.2. Sopra descritto a parole è semplificata la cosa ma per dare un'idea, poi nel manuale ci sono dei riferimenti a diagramma che riporto, e poi entriamo nel dettaglio del codice (pag 140 del manuale)

Transmit Data

The following flow diagram describes the transmit sequence.

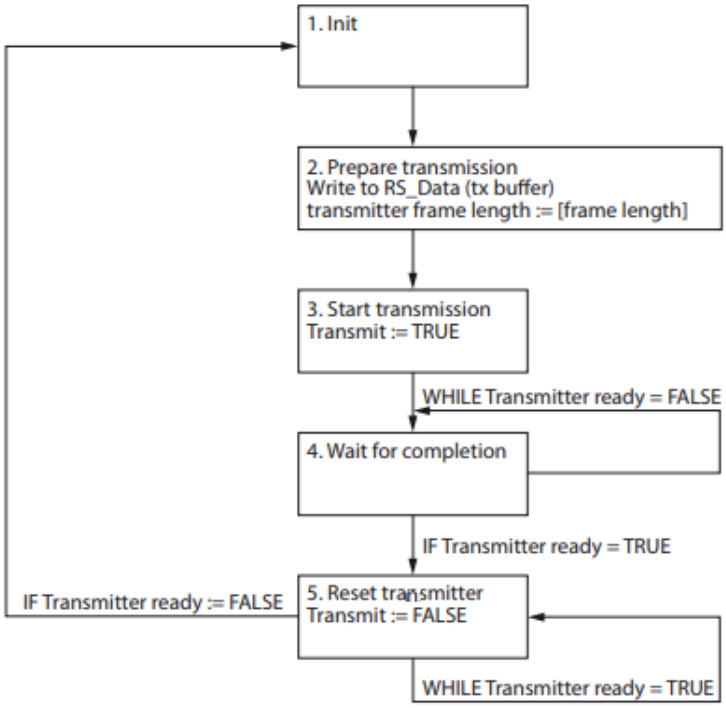


Fig. 72: Transmit sequence

Transmit sequence

Initial state "Transmitter ready" is FALSE (1.).

- Write transmit data (RS_Data) to the transmit buffer (TX buffer) (2.).
- Write the transmit data length in bytes to the process output value "Transmitter frame length" (2.).
- Set the process output value "Transmit" to TRUE (3.).
- Wait until the process input value "Transmitter ready" = TRUE (4.).
- Set the process output value "Transmit" to FALSE (5.).
- Go back to 1 for the next transmit sequence.

Receive Data

The following flow diagram describes the receive sequence.

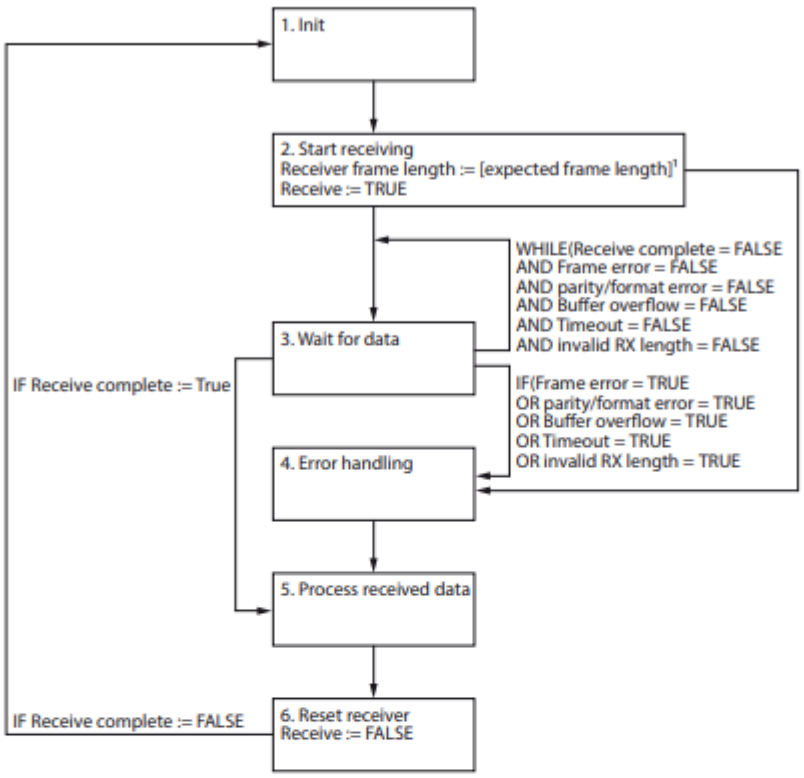


Fig. 73: Receive sequence

Receive sequence

Initial state "Receive complete" is FALSE (1).

- Set the process output value "Receive" to TRUE (starts the receiver) (2.).
- Wait until the process input value "Receive complete" = TRUE or until an error is signaled (3.).
- Perform an error handling. If no error is signaled continue with (5.).
- Read and process the data received from the receive buffer (5.).
- Set the process output value "Receive" to FALSE (stops the receiver) (6.).
- Wait until the process input value "Receive complete" = FALSE.
- Go back to 1 for the next receive sequence.

The following must be observed for receiving data:

- The receiver temporarily has to be deactivated between two transmit sequences (refer to steps 5…8). The duration for the deactivation depends on the update time set and the PLC cycle time. During this time, no data can be received.
- The data reception is limited to 192 bytes per telegram.

2.7. Gestione dei canali Digitali Universali

2.7.1. Essendo canali universali possono essere usati come ingresso o uscita digitale, ma naturalmente saranno linkati a due aree di memoria distinte:

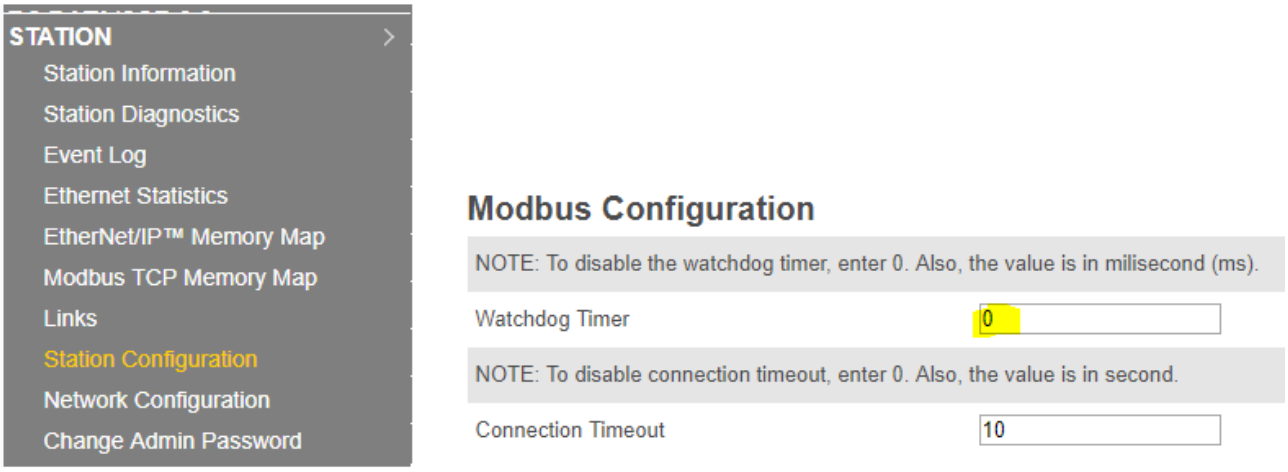
2.7.2. Input

00E8	4	1	1	Intern-S2-2COM-4DXP	DXP 4/Input values/Input value	0 : off 1 : on
00E8	5	1	1	Intern-S2-2COM-4DXP	DXP 5/Input values/Input value	0 : off 1 : on
00E8	6	1	1	Intern-S2-2COM-4DXP	DXP 6/Input values/Input value	0 : off 1 : on
00E8	7	1	1	Intern-S2-2COM-4DXP	DXP 7/Input values/Input value	0 : off 1 : on

2.7.3. Output

08C6	4	1	1	Intern-S2-2COM-4DXP	DXP 4/Output values/Output value	0 : off 1 : on
08C6	5	1	1	Intern-S2-2COM-4DXP	DXP 5/Output values/Output value	0 : off 1 : on
08C6	6	1	1	Intern-S2-2COM-4DXP	DXP 6/Output values/Output value	0 : off 1 : on
08C6	7	1	1	Intern-S2-2COM-4DXP	DXP 7/Output values/Output value	0 : off 1 : on

2.7.4. A seconda della modalità di gestione delle uscite digitali, ovvero se il controllore ciclicamente scrive lo stato aggiornato delle uscite allora si può lasciare il valore di default del watchdog sul modbus TCP , altrimenti se si preferisce gestire ad evento, ovvero cambiare lo stato delle uscite solo quando richiesto allora si deve impostare il watchdog del modbus TCP a 0 per disabilitarlo, altrimenti azzererebbe l'uscita digitale.



3. PTL110S:

3.1. Datasheet :

3.1.1. http://info.bannerengineering.com/cs/groups/public/documents/literature/206183.pdf

3.2. Manuale :

3.2.1. http://info.bannerengineering.com/cs/groups/public/documents/literature/206185.pdf

3.3. Registri Modbus RTU

3.3.1. http://info.bannerengineering.com/cs/groups/public/documents/literature/209995.pdf

3.4. La base di comunicazione è Modbus RTU quindi si deve ragionare a registri, per poter eseguire i vari comandi.

3.4.1. Quindi vi posto un semplice documento che descrive la composizione de byte per i comandi Modbus RTU

3.4.2. http://www.sti.uniurb.it/romanell/Domotica_e_Edifici_Intelligenti/110113-Lez05b-Modbus-DomoticaEdEdificiIntelligenti-Romanelli.pdf

3.4.3. I comandi possibili sono tanti, ma praticamente quelli da utilizzare sono :

- ReadHoldingRegister(Funzione 3)
  - Richiedo la lettura di alcuni dati, quindi mi devo aspettare dei dati in rtirono
- PresetSingleRegister(Funzione6)
  - Scrivo dei dati in un registro specifico, senza aspettarmi dati in ritorno
- PresetMultipleRegister(Funzione16)
  - Scrivo più dati in più registri sepcifici

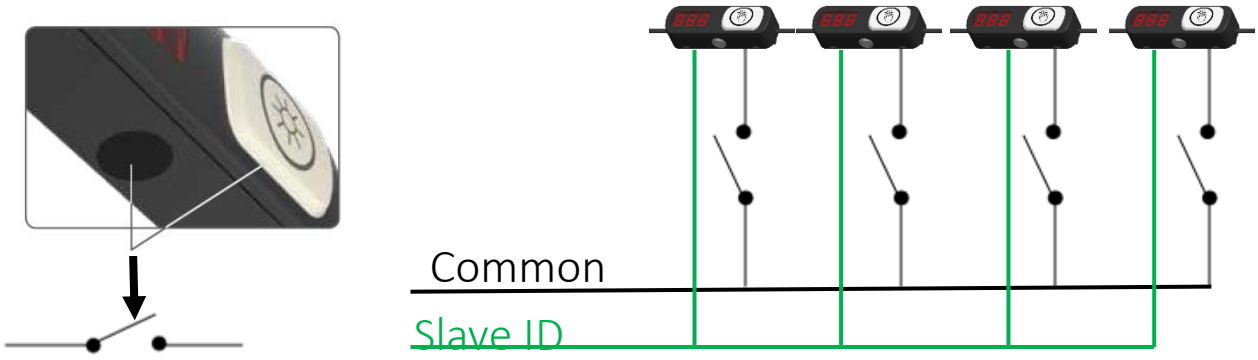
3.4.4. In pratica i comandi seguono le stesse regole del ModbusTCP, ma in questo caso l'accesso ai registri Modbus RTU deve essere costruito per poter essere gestito con il TBEN

3.5. Hardware :

3.5.1. IP54
3.5.2. RGB Inicator / PushButton (14 Color,Animation)
3.5.3. Pig-tail connection Male-Female per facile installazione
3.5.4. Sensore Ottico da 100 mm o 200 mm di range
3.5.5. Display a 3-digit a sette segmenti (max 100 Char, Auo-Scroll, Invert possible)
3.5.6. Possibilità di avere modelli senza alcuni elementi

3.5.7.
3.5.8. Max 246 dispositivi collegati in serie (limite dell'alimentazione degli stessi)
3.5.9. Pick-IQ

- Basato su Modbus RTU, RS485
- Common Slave ID
  - Ogni dispositivo ha due indirizzi, uno specifico e assoluto sulla rete e uno comune definito Common ID uguale per tutti i dispositivi quindi si può interrogare un solo Indirizzo e aspettare il primo che risponde e quindi sui registri dedicati si trova l'indirizzo univoco del dispositivo stesso.



3.6. Filosofia di utilizzo del Pick-IQ:

3.6.1. Il concetto nasce per superare i limiti di simultaneità classici del Modbus RTU, e quindi della difficile scalabilità su grandi numeri.

- Quindi il controllore chiede a tutti tramite il Common ID, il primo che risponde viene gestito, inoltre la richieste rimangono attive finché non vengono gestite

COMMON ID

■ Ask everyone - Who responds first?

SID

■ Finish first conversation, then on to the next

3.6.2. Il Common ID di default è 195, quindi lato controllore continuo ad interrogare lo slave 195, che costantemente non mi risponderà e il controllo darà risposta Timeout, solo quando qualcuno attiverà un nodo allora avverrà la risposta con il TXDone e RXDone, e potrà valutare i registri dedicati al Common ID :

Holding Registers When Common ID is Active

Address without Offset	Address with Offset	Description	Holding Register Representation
7940	7941	Modbus slave ID of active device, same as register 6100	
7941	7942	Device output latch register; values in this register will latch until acknowledged and cleared by the master (either by changing the value in this register or in register 8700) OR will clear after the timeout elapses as defined in register 8812	0 = None triggered 1 = Primary triggered 2 = Secondary triggered 3 = Both triggered
7942	7943	Device output status; values in this register will reflect the real time status of the device's outputs	0 = None triggered 1 = Primary triggered 2 = Secondary triggered 3 = Both triggered

- Dove troveremo L'indirizzo dello slave attivato
- Lo stato del sensore Latch, ovvero una volta premuto/intercettata la ftc o pulsante, rimane attivo finché non cambia lo stato
- Lo stato del sensore in Real time

3.6.3. Quindi saprò quale è lo slave attivato e potrò attuare le strategie necessarie

3.7. Funzionamento PTL110S

3.7.1. Esistono due modalità di funzionamento delle PTL110s

- Base mode
  - In pratica il gestore PC/PLC deve controllare lo stato dei sensori, Luce, Display per ogni singolo evento in base alle proprie necessità (modalità che si sposa poco con i sistemi PickToLight classici)
- State Mode
  - In pratica il PTL è pre-programmato e cambia il suo stato in base agli eventi con la classica modalità di funzionamento:

	No Activation	Activation
Job Input OFF	WAITING	MISPICK
Job Input ON	JOB	ACKNOWLEDGE

3.7.2. I registri per modificare i vari stati sono descritti nel manuale, ma giusto per chiarire il concetto di State mode, vediamo i registri che vanno a regolare il funzionamento di questi 4 stati, che una volta pre-programmati, anche in broadcast per tutti gli slave non si dovrà più agire sugli stessi.

- In pratica ci sono 5 stati e non 4, ovvero ACK può essere attivato dai due sensori, o la fotocellula o dal pulsante e quindi ci sono due reazioni diverse impostabili sempre con strutture di 11 registri
- Waiting State = parte dal registro 6301 per 10 registri
- MissPick State = parte dal registro 6312 per 10 registri
- Job State = parte dal registro 6323 per 10 registri
- Ack State = parte dal registro 6334 per 10 registri
- Ack State Secondario = parte dal registro 6345 per 10 registri

6301	6302	Waiting State: Animation	0 = Off 1 = Steady 2 = Flash 3 = Two Color Flash 4 = Half/Half Top/Bottom 5 = Half/Half Left/Right 6 = Half/Half Rotate 7 = Chase 8 = Intensity Sweep
6302	6303	Waiting State: Color 1	0 = Red 1 = Green 2 = Yellow 3 = Blue 4 = Magenta 5 = Cyan 6 = White 7 = Amber 8 = Rose 9 = Lime Green 10 = Orange 11 = Sky Blue 12 = Violet 13 = Spring Green
6303	6304	Waiting State: Color 2	0 = Red 1 = Green 2 = Yellow 3 = Blue 4 = Magenta 5 = Cyan 6 = White 7 = Amber 8 = Rose 9 = Lime Green 10 = Orange 11 = Sky Blue 12 = Violet 13 = Spring Green
6304	6305	Waiting State: Intensity for color 1	0 = High 1 = Medium 2 = Low 3 = Off
6305	6306	Waiting State: Intensity for color 2	0 = High 1 = Medium 2 = Low 3 = Off
6306	6307	Waiting State: Animation speed	0 = Slow 1 = Standard 2 = Fast
6307	6308	Waiting State: Animation pattern	0 = Normal 1 = Strobe 2 = 3-Pulse 3 = SOS 4 = Random
6308	6309	Waiting State: Animation direction	0 = Clockwise 1 = Counterclockwise
6309	6310	Waiting State: Visual on delay (ms)	0–65535
6310	6311	Waiting State: Visual off delay (ms)	0–65535

3.7.3. Registri da impostare per un ottimale funzionamento sono quelli relativi al setupHW

Holding Registers for Outputs

Use these registers to differentiate sensor outputs or turn them off.

Address without Offset	Address with Offset	Description	Holding Register Representation	Default Value
6000	6001	Touch sensor output (if present)	0 = Disabled 1 = Primary 2 = Secondary	1
6001	6002	Touch sensor on delay (ms)	0–65535	0
6002	6003	Optical sensor output (if present)	0 = Disabled 1 = Primary 2 = Secondary	1
6003	6004	Optical sensor on delay (ms)	0–65535	0

3.7.4. Inoltre abbiamo i registri relativi al Common ID, che influenza il comportamento del sistema Pick-IQ

- In particolare i tempi relativi allo stato di Latch dei sensori

Common ID Configuration Holding Registers

Address without Offset	Address with Offset	Description	Holding Register Representation	Default Value
8810	8811	Common ID	1–247	195
8811	8812	Global on delay that applies to both inputs (touch and optical sensor) (stacks on top of on delays in registers 6001 and 6003) (ms)	0–65535 (65535 value is infinite)	0
8812	8813	Latch timeout for 7941 (ms)	0–65535 (65535 value is infinite)	1000
8813	8814	Minimum output on time for register 7942, off delay (ms)	0–65535 (65535 value is infinite)	1000

3.7.5. Il registro che rappresenta il JOB Input è il 8700 (0=OFF, 1=ON attivato)

3.8. Costruzione comandi Modbus RTU

3.8.1. ReadingHoldingRegister (Funzione 3)

Domanda

Oltre all'indirizzo dello slave e al codice funzione (03) il messaggio contiene l'indirizzo di partenza (starting Address) espresso su due byte e il numero di word da leggere anch'esso su due byte. Il numero massimo dì word che possono essere lette è 125. La numerazione degli indirizzi parte da zero (word1= 0) per il MODBUS, da uno (word1 =1)per il JBUS

Esempio: Richiesta di lettura dallo slave 25 dei registri da 069 a 0071.

ADDR	**FUNC**	**DATA start Addr HI**	**DATA start Addr LO**	**DATA bit # HI**	**DATA bit # LO**	**CRC HI**	***CRC* LO**
19	03	00	44	00	03	46	06

Risposta

Oltre all'indirizzo dello slave e al codice funzione (03) il messaggio comprende un carattere che contiene il numero di byte di dati e i caratteri contenenti i dati.
I registri richiedono due byte ciascuno, il primo dei quali contiene la parte più significativa.

Esempio: Risposta alla richiesta sopra riportata.

ADDR	**FUNC**	**DATA byte count**	**DATA byte 69 HI**	**DATA byte 69 LO**	**DATA byte 70 HI**	**DATA byte 70 LO**	**DATA byte 71 HI**	**DATA byte 71 LO**	**CRC HI**	***CRC* LO**
19	03	06	02	2B	00	00	00	64	AF	7A

3.8.2. Come si vede si ragiona in Byte e quindi nella nostra costruzione dei dati bisognerà ragionare così, e sempre per comodità spesso si ragiona in esadecimale

3.8.3. In pratica si compone la prima richiesta

- ADDR = Indirizzo dello slave a cui si vuole accedere
- FUNC = 3 = reading holding register
- DATA Start = L'indirizzo del registro a cui si vuole accedere splittato su due byte
- DATA = Quanti registri si vuole recuperare splitatto su due byte
- CRC = CRC 16 dei byte precedenti (Il calcolo del CRC si trovano varie esempi via funzione ricorsiva di calcolo o via tabelle già costruite)

3.8.4. E si aspetta una risposta formattata in una certa modalità

- ADDR = Indirizzo dello slave che ha risposto
- FUNC = funzione eseguita
- DATA Byte Count = Quanti byte di dati si troveranno nella risposta
- DATA n BYTE = registri splitati in byte (6 byte = 3 word di dati perché avevamo chiesto 3 registri)
- CRC = splittato su due byte, CRC di controllo che può anche essere ignorato

3.8.5. PresetSingleRegister (Funzione 6)

3.6 Preset Single Register (06)

Questa funzione permette di impostare il valore di un singolo registro a 16 bit.
Il modo broadcast è permesso.

Domanda

Oltre all'indirizzo dello slave e al codice funzione (06) il messaggio contiene l'indirizzo della variabile espresso su due byte e il valore che deve essere assegnato.
La numerazione degli indirizzi parte da zero (word1 = 0) per il MODBUS, da uno (word1 = 1) per il JBUS.

Esempio: Richiesta di forzare 928 sullo slave 35 all'indirizzo 26.

ADDR	FUNC	DATA bit # HI	DATA bit # LO	DATA Word HI	DATA Word LO	CRC HI	*CRC* LO
23	06	00	19	03	A0	5E	07

Risposta

La risposta consiste nel ritrasmettere il messaggio ricevuto dopo che la variabile è stata modificata.

Esempio: Risposta alla richiesta sopra riportata.

ADDR	FUNC	DATA bit # HI	DATA bit # LO	DATA Word HI	DATA Word LO	CRC HI	*CRC* LO
23	06	00	19	03	A0	5E	07

3.8.6. In pratica si può fare la richiesta di trasmissione senza aspettare alcuna risposta, anche se per essere sicuri che sia eseguita bisognerebbe valutare la risposta

3.8.7. Nella scrittura si può indicare l'indirizzo specifico dello slave oppure mandare una messaggio di scrittura in broadcast , ovvero a tutti gli slave mettendo come indirizzo "0", ovviamente non si avrà risposta

3.8.8. Per la formattazione vale quanto descritto per il ReadingHoldingRegister

- DATA Bit = Indirizzo del registro da modificare sempre splittati su due byte
- DARA Word = Dati da inserire sempre splittati su due byte

4. Integrazione del sistema (esempio con codice)

4.1. Partendo da tutte le info di cui sopra diamo un esempio di integrazione su un applicativo ad alto livello.

4.2. Primo step Comunicazione Modbus TCP con il TBEN, sfruttando una libreria Modbus

```
from pyModbusTCP.client import ModbusClient
import struct
import math
import time
import array
from array import *
import time
from random import randint
import sys
import socket
import serial
import minimalmodbus
import numpy as np
```

4.3. Quindi la sua gestione aprendo il socket di comunicazione ed eventuali suoi errori

```
##############################################
#Define Data Init
if True :
    SERVER_HOST = "192.168.1.112"
    SERVER_PORT = 502

    c = ModbusClient()

    # uncomment this line to see debug message
    # c.debug(True)

    # define modbus server host, port
    c.host(SERVER_HOST)
    c.port(SERVER_PORT)
```

```
###Controllo che sia aperta la connessione
if not c.is_open():
    if not c.open():
        print("unable to connect to " + SERVER_HOST + ":" + str(SERVER_PORT))
        c.close()
        quit()
```

4.4. Useremo le funzioni della libreria per leggere i dati o per scriverne :

4.4.1. Ad esempio scrivendo 3 registri con valore 0, nei registri 16#0800, 16#0801, 16#0802, ovvero i comandi della porta COM del TBEN

```
_TbenRegCommandIniz = 0x0800
```

```
data2 = [0, 0, 0]
regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data2)
```

4.4.2. Oppure lettura dei 12 registri dei dati estratti, quindi estraendo i valori dal registro 16#0003 al 16#000E, ovvero quelli dei dati Input della COM0

```
_TbenRegStatusIniz = 0x0000
```

```
data = c.read_holding_registers(_TbenRegStatusIniz+3, 12)
#print("dati estratti", data)
```

4.5. Quindi useremo un semplice WHILE sempre attivo per creare una sorta di ciclo di esecuzione e poter valutare lo stato dei segnali che arrivano dalla porta COM che sono asincroni rispetto alla comunicazione Modbus TCP.

```
while True:


    tBenreadStatuBIt(c)

    txDone=tBenreadStatuBIt(c)[0]
    RxDone=tBenreadStatuBIt(c)[1]
    RxTimeout =tBenreadStatuBIt(c)[2]

    #print("txDone", type(txDone))
    #print("RxDone", RxDone)
    #print("RxTimeout", RxTimeout)
    #print("step",step)




    if step==1:...
```

4.6. E si può creare una funzione di recupero dei dati relativi agli stati, ovvero per i primi 2 registri in ingresso ovvero quelli dei bit relativi alla porta COM, che dovranno essere aggiornati ciclicamente per essere usati come condizioni di avanzamento delle logiche

```
def tBenreadStatuBIt(mb):

    """
    leggo i dati relativi al tben, ovvero i dati in ingresso
    Il modulo tben ==> la porta COM va impostata come COM serial RS485 advanced
    Quindi i vanno gestiti i bit di comunicazione
    """

    regs = mb.read_holding_registers(_TbenRegStatusIniz, _TbenRegStsusLengh)

    #Quindi splitto i vari dati
    regs0_statusBit = regs[0]
    regs1_frameLenght = regs[1]
    #regs2_DataWord = bytes(regs[3:15])


    #Creo bit per gli status
    _TbenStatusTxReady     = regs0_statusBit & 0b00000001
    _TbenStatusRXComplete  = regs0_statusBit & 0b00000010
    _TbenStatusRxTimeout   = regs0_statusBit & 0b00100000

    #print("####StatusBit tutta la word" , regs0_statusBit)
    #print("####_TbenStatusTxReady"      , _TbenStatusTxReady)
    #print("####_TbenStatusRXComplete"   , _TbenStatusRXComplete)
    #print("####_TbenStatusRxTimeout"    , _TbenStatusRxTimeout)
    #print("################################################################################", regs1_frameLenght)
    #print("################################################################################", regs2_DataWord)


    return [_TbenStatusTxReady, _TbenStatusRXComplete, _TbenStatusRxTimeout ]
```

4.7. A questo punto si rendono necessarie delle funzioni per comporre il telegramma da mandare sulla rete Modbus RTU agli slave.

```
def ModbusRTUWiteSingleReg(AddrNode, StartReg, Value):

    _TbenRegAddrReg_0 = splitIntToBytes(StartReg)[1]
    _TbenRegAddrReg_1 = splitIntToBytes(StartReg)[0]
    _TbenRegValue_0   = splitIntToBytes(Value)[1]
    _TbenRegValue_1   = splitIntToBytes(Value)[0]

    Func=6

    data = bytearray()
    data.append(AddrNode)
    data.append(Func)
    data.append(_TbenRegAddrReg_0)
    data.append(_TbenRegAddrReg_1)
    data.append(_TbenRegValue_0)
    data.append(_TbenRegValue_1)
    #print("###Stringa di byte", data)
    crc=int(crc16(data))
    #print("###CRC", crc)
    crc1=splitIntToBytes(crc)[0]
    crc2=splitIntToBytes(crc)[1]
    data.append(crc1)
    data.append(crc2)
    #print("###totale", data)


    sendData_00 = data[1]*256+data[0]
    sendData_01 = data[3]*256+data[2]
    sendData_02 = data[5]*256+data[4]
    sendData_03 = data[7]*256+data[6]

    #print("prima word   ", sendData_00 )
    #print("seconda word ", sendData_01 )
    #print("terza word   ", sendData_02 )
    #print("quarta word  ", sendData_03 )

    data= [sendData_00, sendData_01, sendData_02, sendData_03]

    regsDataRTU = c.write_multiple_registers(_TbenRegCommandIniz+3,data)
```

4.7.1. Dove passiamo come parametri passiamo: Indirizzo del nodo, indirizzo del registro iniziale e il valore.

4.7.2. Usiamo la funzione 6 (PresetSingleRegister)

4.7.3. Avendo deciso di scrivere un solo registro alla volta resta più facile la gestione in quanto la composizione è statica, ma possiamo inviare questo valore in broadcast mettendo come indirizzo “0”.

4.7.4. Il protocollo essendo a base Byte si dovrà comporre la stringa a byte e quindi aggruppandoli a word, twistando i byte.

4.7.5. Per il calcolo del CRC16 usato una funzione che in questo caso sfrutta NumPy ma se ne trovano diverse anche partendo dalle librerie Modbus RTU, che spesso sono integrate a quelle Modbus TCP

```
def crc16(data: bytes):
    '''
    CRC-16-ModBus Algorithm
    '''
    data = bytearray(data)
    poly = 0xA001
    crc = 0xFFFF
    for b in data:
        crc ^= (0xFF & b)
        for _ in range(0, 8):
            if (crc & 0x0001):
                crc = ((crc >> 1) & 0xFFFF) ^ poly
            else:
                crc = ((crc >> 1) & 0xFFFF)

    return np.uint16(crc)
```

4.7.6. Quindi i dati vengono spediti con i comandi Modbus TCP.

4.8. Inoltre servirà una funzione per comporre il telegramma per richiedere i dati sul modbus RTU agli slave

```
def ModbusRTUReadHoldingReg(AddrNode, StartReg, Lenght):

    _TbenRegstart_0 = splitIntToBytes(StartReg)[1]
    _TbenRegstart_1 = splitIntToBytes(StartReg)[0]
    _TbenRegLenght_0   = splitIntToBytes(Lenght)[1]
    _TbenRegLenght_1   = splitIntToBytes(Lenght)[0]

    Func=3

    data = bytearray()
    data.append(AddrNode)
    data.append(Func)
    data.append(_TbenRegstart_0)
    data.append(_TbenRegstart_1)
    data.append(_TbenRegLenght_0)
    data.append(_TbenRegLenght_1)
    #print("###Stringa di byte", data)
    crc=int(crc16(data))
    #print("###CRC", crc)
    crc1=splitIntToBytes(crc)[0]
    crc2=splitIntToBytes(crc)[1]
    data.append(crc1)
    data.append(crc2)
    #print("###totale", data)


    sendData_00 = data[1]*256+data[0]
    sendData_01 = data[3]*256+data[2]
    sendData_02 = data[5]*256+data[4]
    sendData_03 = data[7]*256+data[6]

    #print("prima word   ", sendData_00 )
    #print("seconda word ", sendData_01 )
    #print("terza word   ", sendData_02 )
    #print("quarta word  ", sendData_03 )

    data= [sendData_00, sendData_01, sendData_02, sendData_03]


    regsDataRTU = c.write_multiple_registers(_TbenRegCommandIniz+3,data)
```

4.8.1. Dove passiamo, indirizzo del nodo, indirizzo del primo registro, e numero di registri da leggere

4.8.2. Usiamo Funzione 3, ovvero ReadingHoldingRegister

4.8.3. Componiamo sempre a Byte, calcoliamo il CRC16 e twistiamo i byte, quindi li mandiamo nell'area Dati Output della COM0 (16#0803)

4.9. E in ultimo la funzione di decodifica dei dati recuperati a seguito della richiesta di lettura dei dati tramite ReadingHoldingRegister (Tutte le funzioni si potrebbero ottimizzare ma sono state messe per esteso per maggiore comprensione)

```
def ModbusRTUDecodeReadHolding(Data):


    #ragionando a BYte al risposta è composta da
    #(Addr+FUnc) + (ByteCOunt+data0Hi) + (data0Lo+data1Hi)  + (data1Lo+data2Hi) + (data2Lo+data3Hi)  + (data3Lo+data4Hi) + (dat

    _Addr       = splitIntToBytes(Data[0])[0]
    _Func       = splitIntToBytes(Data[0])[1]
    _ByteCount  = splitIntToBytes(Data[1])[0]

    _data0_hi   = splitIntToBytes(Data[1])[1]
    _data0_lo   = splitIntToBytes(Data[2])[0]

    _data1_hi   = splitIntToBytes(Data[2])[1]
    _data1_lo   = splitIntToBytes(Data[3])[0]

    _data2_hi   = splitIntToBytes(Data[3])[1]
    _data2_lo   = splitIntToBytes(Data[4])[0]

    _data3_hi   = splitIntToBytes(Data[4])[1]
    _data3_lo   = splitIntToBytes(Data[5])[0]

    _data4_hi   = splitIntToBytes(Data[5])[1]
    _data4_lo   = splitIntToBytes(Data[6])[0]

    _data5_hi   = splitIntToBytes(Data[6])[1]
    _data5_lo   = splitIntToBytes(Data[7])[0]

    _data6_hi   = splitIntToBytes(Data[7])[1]
    _data6_lo   = splitIntToBytes(Data[8])[0]

    _data7_hi   = splitIntToBytes(Data[8])[1]
    _data7_lo   = splitIntToBytes(Data[9])[0]

    _data8_hi   = splitIntToBytes(Data[9])[1]
    _data8_lo   = splitIntToBytes(Data[10])[0]

    _data9_hi   = splitIntToBytes(Data[10])[1]
    _data9_lo   = splitIntToBytes(Data[11])[0]


    #rebuild the value in order

    #print("Indirizzo   ", _Addr )
    #print("funzione    ", _Func )
    #print("Numero byte ", _ByteCount )

    _DataWord_0 = _data0_hi*256 + _data0_lo
    _DataWord_1 = _data1_hi*256 + _data1_lo
    _DataWord_2 = _data2_hi*256 + _data2_lo
    _DataWord_3 = _data3_hi*256 + _data3_lo
    _DataWord_4 = _data4_hi*256 + _data4_lo
    _DataWord_5 = _data5_hi*256 + _data5_lo
    _DataWord_6 = _data6_hi*256 + _data6_lo
    _DataWord_7 = _data7_hi*256 + _data7_lo
    _DataWord_8 = _data8_hi*256 + _data8_lo
    _DataWord_9 = _data9_hi*256 + _data9_lo


    #if _ByteCount>=2 :  print("1 word   ", _DataWord_0 )
    #if _ByteCount>=4 :  print("2 word   ", _DataWord_1 )
    #if _ByteCount>=6 :  print("3 word   ", _DataWord_2 )
    #if _ByteCount>=8 :  print("4 word   ", _DataWord_3 )
    #if _ByteCount>=10 : print("5 word   ", _DataWord_4 )
    #if _ByteCount>=12 : print("6 word   ", _DataWord_5 )



    data= [_Addr, _ByteCount, _DataWord_0, _DataWord_1, _DataWord_2, _DataWord_3, _DataWord_4, _DataWord_5, _DataWord_6, _DataW
```

4.9.1. Ricevo in ingresso l'array/lista di dati e li interpreto decodificandoli, in base alla risposta che si ottiene sulla ReadingHoldingRegister, che come si vede ha byte dispari, e quindi vanno sistemati i byte a dovere per poterli interpretare in maniera corretta

4.10. Ora si possono costruire tutte le logiche necessarie, o anche ulteriori funzioni per ottimizzare gli oggetti di utilizzo. All'atto pratico andremo a simulare un simil PickToLight :

4.10.1. Attiviamo tutti gli slave (usando il boradcast)

4.10.2. Quando viene intercettata la ftc di uno qualsiasi viene scritto a monitor il numero del nodo e vengono disattivati

4.10.3. Avendo due nodi, quando entrambi sono stati intercettati si riattivano entrambi.

4.11. Le PTL sono già state programmate nei vari stati, e impostato in modalità di funzionamento State-mode:

4.11.1. Prepariamo i dati, scrivendo 1 nel registro 8700 (Device Job State),nell'area di memoria di DATI-Out del TBEN

```
_TbenCmd_Tx    = 0x0001
_TbneCmd_Rx    = 0x0002
_TbenCmd_TxRx  = 0x0003
```

```
_Ptl110Reg_StateMode = 8700
```

```
399         if (step==10):
400             #attivo tutti i PTL110s
401             ModbusRTUWiteSingleReg(0,_Ptl110Reg_StateMode,1)
402             data = [_TbenCmd_Tx, 8, 8]
403             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
404             step=11
405             clean=False
406             riparti= False
407             resetted=[]
```

4.11.2. Usando la funzione ModbusRTUSingleReg, andiamo a scrivere su tutti i registri mettendo come indirizzo dello slave 0 (Broadcast), mettiamo l'indirizzo del registro, ovvero 8700, e mettiamo il valore 1, ovvero attiviamo lo stato di JOB (all'interno della funzione c'è già la chiamata alla scrittura dei dati su Modbus TCP)

4.11.3. Quindi prepariamo una lista/array con i dati di attivazione dei bit di comando della Porta COM, ovvero la word di comando con il valore 1,, ovvero primo bit attivo, e anche la lunghezza dei byte da trasmettere (8), e quelli che ci aspettiamo di ricevere (8) anche se in questo caso non aspettiamo nessuna risposta

4.11.4. Quindi con la funzione WriteMultipleRegister scriviamo questi dati con Modbus TCP sull'area di memoria dei comandi della COM0.

4.11.5. Quindi una volta eseguite le prime 3 righe ci troveremo con la luce di JOB Light attiva su entrambi i nodi.

4.12. Tenendo conto che il ciclo While sta girando sempre , andiamo allo step successivo, ovvero quello dove attendiamo la risposta asincrona della porta COM0, che ha terminato la Tx.

```
409         if (step==11) & (txDone==1) :
410             #abbasso i bit di comando
411             data = [0, 0, 0]
412             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
413             step=12
```

4.12.1. Quindi entro nello step successivo , ma solo quando il bit di stato TxDone è alto, ovvero la porta COM0 ci informa che ha eseguito il comando.

4.12.2. Quindi il comando è stato eseguito e devo abbassare i bit di comando che avevo attivato, quindi metto a 0 la word di comando e anche la lunghezza dei frame da trasmettere-ricevere

4.12.3. Con la solita chiamata alla scrittura dei dati su Modbus TCP completo l'operazione

4.13. A questo punto entra in gioco la modalità del PICK-IQ, dove andiamo a chiedere costantemente i dati sul COMMON-ID, ovvero indirizzo 195, dove ci aspettiamo una risposta di timeout sempre, e solo quando uno slave viene attivato avremo una risposta con un RXDone e quindi sapremo che si è attivato uno slave

```
415         if (step==12):
416             #richiedo i dati con request Holding register
417             numRegToBeRead = 3
418             ModbusRTUReadHoldingReg(195,7940,numRegToBeRead)
419             data = [_TbenCmd_TxRx, 8, (5+(2*numRegToBeRead))]
420             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
421             step=13
```

4.13.1. Dove andiamo a chiedere 3 registri, ovvero quelli relativi al Common-ID 7940-7941-7942, quindi prepariamo il frame di byte per questa richiesta sul Modbus RTU e lo scriviamo nei Data-Out della COM0

4.13.2. Prepariamo la word di comandi, con il comando di RX e TX, e diamo le lunghezze dei frame, e le scriviamo come al solito nell'area di memoria dei Comandi della COm0 del TBEN.

4.14. Quindi ci aspettiamo di ricevere il TxDone e dopo poco il RxDone o Timeout.

```
423         if (step==13) & (txDone==1):
424             step=14
425
426         if (step==14) & (RxDone==2):
427             data = c.read_holding_registers(_TbenRegStatusIniz+3, 12)
428             #print("dati estratti", data)
429             dataDec=ModbusRTUDecodeReadHolding(data)
430             step=15
431             #time.sleep(5)
432             clean=True
```

4.14.1. Quindi appena riceviamo il TxDone andiamo oltre,

4.14.2. Se riceviamo il RxDone allora vuol dire che qualcuno ha attivato un sensore del PTL110S, e ha risposto alla chiamata e ha popolato i dati nell'area di memoria DATA-INPUT della COM0, quindi mi prendo tutti i 12 registri dei dati

4.14.3. E poi li vado a decodificare con la funzione apposita.

4.15. In contrapposizione se nessuno ha attivato una PTL arriverà il bit di Timeout

```
435         if (step==14) & (RxDone==0) & (RxTimeout==32) :
436             data = [0, 0, 0]
437             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
438             step=15
```

4.15.1. E quindi andiamo a riportare i bit di comando a zero.

4.16. Quindi che abbiamo ricevuto un RxDone o TImeout dobbiamo pulire i comandi, in quanto essi lavorano sui fronti di salita.

```
440         if (step==15) :
441             #abbasso i bit di comando
442             data = [0, 0, 0]
443             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
444             if riparti== True :
445                 step=10
446                 time.sleep(1)
447
448             else :
449                 if clean==True :
450                     step=20
451                 else:
452                     step=12
```

4.17. Quindi il PTL110s che è stato identificato da un azione di un operatore, viene disattivato il JOB, e riportato in Waiting-State

```
456         if (step==20):
457             #disattivo il ptl110selezionato
458             clean=False
459             addr = dataDec[2]
460             ModbusRTUWiteSingleReg(addr,_Ptl110Reg_StateMode,0)
461             data = [_TbenCmd_Tx, 8, 8]
462             regCommRTU = c.write_multiple_registers(_TbenRegCommandIniz, data)
463             step=15
464             resetted.append(addr)
465             print("azzerati", resetted)
466             time.sleep(0.2)
467             if len(resetted) >=2 :
468                 resetted=[]
469                 riparti = True
470                 step=15
```

4.17.1. Il numero del nodo era stato salvato durante la decodifica dei dati, quindi viene usato per accedere direttamente allo slave stesso, portando il registro 8700 a zero

4.17.2. Quindi rialzati i bit di comando per attivare la trasmissione sul ModbusRTU

4.17.3. In seguito i Bit verranno azzerati come al solito sulla ricezione del bit di stato TxDone.

4.18. Conclusioni sull'esempio :

4.18.1. Esempio pratico, molto semplice che non va a coprire tutte le possibilità, ma serve per illustrare la modalità di scambio dati fra ModbusTCP verso il TBEN, quindi come preparare i dati per gli Slave e come triggerare le azioni sul modbusRTU.