DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Implementazione del protocollo Modbus su microcontrollori

Relatore
Prof. Buso Simone

Laureando
Agatea Alberto

ANNO ACCADEMICO 2023-2024

Data di laurea 19/11/2024

Indice

Indice 3

Sommario 5

1. Capitolo 1: **Introduzione** 7
   1.1. Cos'è il protocollo di comunicazione Modbus 7
   1.2. Perché è così diffuso 7
2. Capitolo 2: **Protocollo Modbus** 9
   2.1. Descrizione generale del protocollo 9
   2.2. Com'è strutturato il protocollo 10
      2.2.1 Breve excursus sul modello OSI 10
      2.2.2 Modbus Physical Layer (RS485) 12
      2.2.3 Modbus DataLink Layer 15
      2.2.4 Modbus Application Layer 19
   2.3. Funzioni 20
   2.4. Gestione degli Errori 23
      2.4.1. Eccezione da parte dello slave 24
      2.4.2. Calcolo del CRC 25
3. Capitolo 3: **Implementazione Modbus RTU su RS485 con STM32F103 e STM32F334** 27
   3.1. Interfacciamento STM32 con RS485 27
   3.2. Configurazione STM32F103 28
   3.3. Creazione del progetto in uVision 5 32
   3.4. Configurazione STM32F334 40
4. Capitolo 4: **Conclusioni** 43
5. **Appendice** 44
6. **Bibliografia/sitografia** 49

Sommario

In questo documento viene esposta una breve panoramica del protocollo di comunicazione seriale Modbus, per poi vedere più nel dettaglio il protocollo Modbus RTU.
Dopo aver affrontato, a livello teorico, quali siano le caratteristiche distintive di tale protocollo si spiega in che modo si possa realizzarne un'implementazione tramite il software uVision 5 a seguito di un'opportuna configurazione del microcontrollore che assumerà il ruolo del Master, ovvero STM32F103.
Una volta realizzato il codice, si osserva il messaggio inviato sulla periferica USART tramite il simulatore di uVision. Vengono poi fatte delle considerazioni al riguardo e si configura un microcontrollore di livello superiore, STM32F334, che viene messo a confronto con il precedente.

Capitolo 1

Introduzione

1.1 Cos'è il protocollo Modbus

Modbus è un protocollo di comunicazione dati di tipo client/server creato da Modicon (da cui "Mod"-Bus) nel 1979, originariamente per mettere in comunicazione su base seriale i propri PLC. Tuttavia negli anni è diventato uno standard de facto nella comunicazione, soprattutto in ambito industriale, fra dispositivi elettronici come sensori, attuatori, controller e sistemi di supervisione.

1.2 Perché è così diffuso

Nonostante l'azienda produttrice, Modicon, faccia ora parte del gruppo Schneider Electric il protocollo Modbus ha avuto modo di diffondersi ampiamente nel settore dell'automazione soprattutto per la "strategia di vendita" utilizzata dall'azienda.
Si tratta infatti di un protocollo con una licenza di tipo "royalties-free", cioè con limitate restrizioni sul suo utilizzo. Non imponendo vincoli a livello di implementazione, si è verificata una rapida diffusione in tutto il mercato industriale, non più solo come comunicazione seriale, ma per esempio anche su base Ethernet.
Ai vantaggi si aggiungono anche la semplicità in sé del protocollo, e le risorse hardware/software esigue garantiscono l'applicazione anche su microprocessori economici.
Inoltre, il protocollo si presta anche ad implementazioni parziali dei comandi con la possibilità di utilizzare dei comandi custom da parte del produttore.

Capitolo 2

Protocollo Modbus

2.1 Descrizione generale del protocollo Modbus

Il protocollo Modbus è basato su bus di campo, permettendo lo scambio di informazioni tra apparecchiature posizionate a notevoli distanze tra loro ed essendo un protocollo *open* dà la possibilità di interconnettere dispositivi commerciali realizzati da diversi produttori e facilmente sostituibili tra loro. Il fatto di "unificare" l'interconnessione attraverso una stessa rete permette di ridurre anche i tempi di un'eventuale manutenzione dell'impianto.

Esistono diversi tipi di Modbus, ciascuno progettato per differenti esigenze ed ambienti di comunicazione. Ne viene fatta una breve distinzione di seguito in base alle modalità di connessione:

- Tramite *comunicazione seriale RS232 o RS485*:
  - Modbus RTU
  - Modbus ASCII

  i quali loro volta si distinguono per le codifiche utilizzate.

- Tramite *comunicazione Ethernet*:
  - Modbus TCP/IP, il quale sfrutta la suite di protocolli internet TCP/IP e si tratta dunque di una versione senza pacchetti aggiuntivi per il checksum.

  - Modbus UDP, simile al TCP/IP ma che si basa appunto sul protocollo UDP (UserDataProtocol) il quale non prevede il controllo di flusso e la gestione degli errori, aumentando la velocità di comunicazione ma riducendo l'affidabilità.

  - Modbus Plus, diverso dai precedenti essendo una rete di tipo peer-to-peer e non master/slave (è una versione proprietaria di Modbus sviluppata da Schneider Electronic).

Nella tesi verrà analizzata principalmente la comunicazione seriale tramite **Modbus RTU**.

2.2 Com’è strutturato il protocollo Modbus

Il tipo di connessione (seriale, nel caso citato sopra) definisce il Physical Layer del protocollo ossia l’hardware sul quale avviene lo scambio di informazioni.
Un protocollo di comunicazione è generalmente composto da diversi livelli o strati. Il modello usato per rappresentare i protocolli Modbus RTU e Modbus ASCII è il modello OSI (Open System Interconnection), che divide il protocollo di comunicazione in 7 strati, ciascuno con funzioni differenti.

2.2.1 Breve excursus sul modello OSI

Di seguito viene quindi data una breve spiegazione della struttura generale di un protocollo di comunicazione, cioè dei suoi layer, per poi passare concretamente a parlare del protocollo Modbus.
L’architettura generale del sistema OSI si basa essenzialmente su:

1) I sistemi che contengono le applicazioni
2) I processi applicativi per lo scambio di informazioni
3) Le connessioni che permettono lo scambio di informazioni

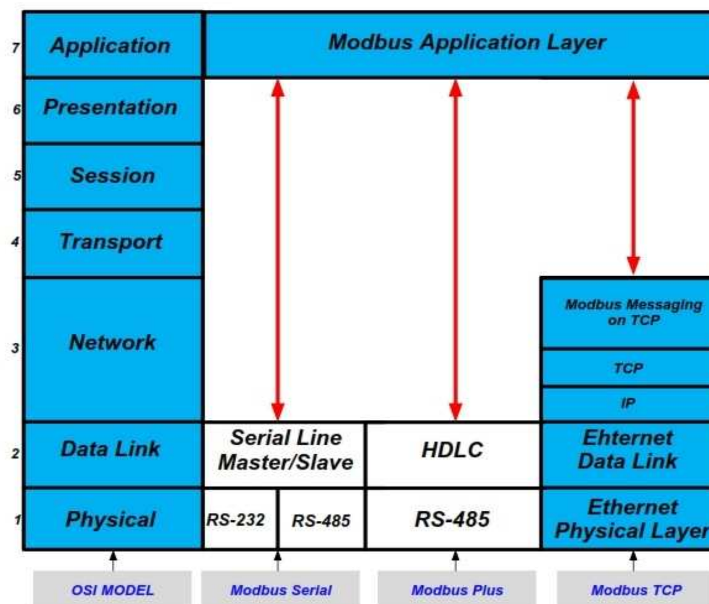

Fig. 2.1: Modello OSI

A loro volta i livelli sono divisi in 2 classi:
I primi 3 fanno parte della classe denominata *protocolli di rete o di livello inferiore*, riguardano la rete di comunicazione ed il loro compito è di far in modo che i dati arrivino al destinatario in maniera corretta.
I 4 livelli superiori invece si assicurano che i dati, una volta giunti a destinazione, possano essere correttamente interpretati.

Il **livello fisico** riguarda la trasmissione dei bit attraverso il canale di comunicazione e specifica, dunque, le modalità di trasmissione dei bit e le caratteristiche dei segnali che servono per trasmetterli. Nel caso della comunicazione seriale ci sono diversi tipi di standard a cui uniformare il *Physical Layer*, fra cui: RS232, RS485, RS422, SPI, I2C, CAN, UART. Nei riguardi di questa tesi verrà data particolare attenzione alla comunicazione seriale asincrona con RS485.

Il **livello di data link o collegamento dati** organizza la struttura delle informazioni trasmesse dividendole in frame e si occupa poi che tali frame vengano inviati in maniera opportuna.

Per l'implementazione trattata nel seguito si parla di collegamento seriale, del quale esistono diversi modelli a seconda dell'applicazione e delle risorse disponibili. Per citarne alcuni fra i più conosciuti vi sono: P2P, Master/Slave, Multi-drop, Broadcast, Token passing, Point-to-Point, Full Duplex, Half Duplex.
Nel caso del Modbus si tratta di modello Master/Slave dove un dispositivo centrale (il Master appunto) controlla la comunicazione e gestisce la trasmissione dei dati dati tra i dispositivi "subordinati" ovvero gli Slave, i quali non possono comunicare fra loro ed eseguono solo le richieste che arrivano dal master. Possiamo quindi dire che si tratti di una comunicazione unidirezionale.

Il **livello di rete** si occupa di gestire l'indirizzamento dei messaggi attraverso la rete scelta dal livello di trasporto.

Il **livello di trasporto** ha lo scopo di fornire un trasporto dati affidabile e per fare ciò effettua un controllo end to end dei dati per prevenire errori.

Il **livello di sessione** fornisce ai programmi applicativi un insieme di funzioni necessarie per la gestione dei dati ed il loro trasferimento.

Il **livello di presentazione** ha il compito di consentire la corretta interpretazione dei dati scambiati tra due DTE indipendentemente dai codici e dai formati e da tutte le altre convenzioni impiegate da ciascun sistema.

Il **livello di applicazione** contiene i programmi utente o programmi applicativi che consentono all'utente di svolgere le sue attività in rete. Nel nostro caso l'application layer corrisponde al Modbus.

Il protocollo di comunicazione Modbus seriale occupa solamente 3 layer: Application Layer, DataLink Layer e il Physical Layer.
Possiamo quindi dare la seguente descrizione generale, ma concisa di questo protocollo:

"Modbus è un protocollo di messaggistica a livello applicativo, posizionata al livello 7 del modello OSI, che fornisce una comunicazione client/server tra dispositivi collegati fra loro tramite tipi di bus diversi"

2.2.2 Modbus Physical Layer (RS485)

Come precedentemente citato in questo layer viene definito il modo con cui vengono scambiati i dati. La comunicazione tramite standard RS485 può avvenire tramite 2 fili o 4 fili, e di seguito vengono riportate le specifiche per il caso bifilare dato che è maggiormente utilizzato.

Essa è realizzata con un cavo bifilare che mette in parallelo tutti i dispositivi connessi in rete: trattandosi di Modbus RTU, ci può essere un solo master nella rete mentre il numero di slave è maggiore o uguale ad 1.

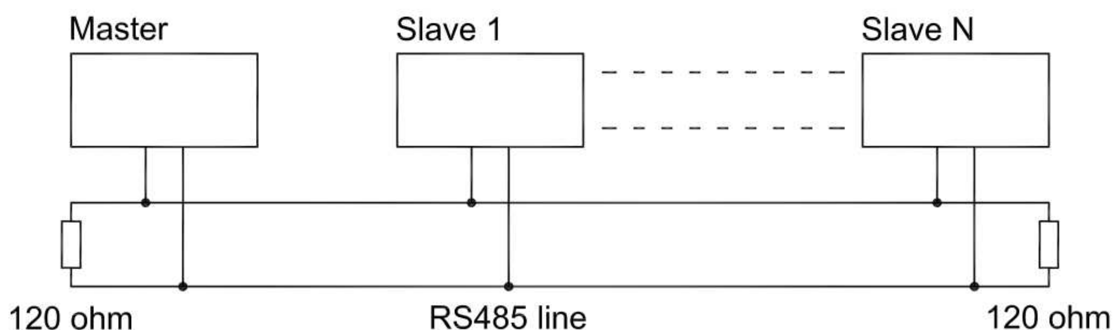

Fig. 2.2: Comunicazione Master e Slave

La linea bifilare garantisce una buona velocità di comunicazione e una notevole immunità ai disturbi di tipo elettromagnetico.

Per trasmettere i bit da un dispositivo ad un altro si applica una differenza di potenziale ad uno dei capi dei fili: la polarità di tale tensione cambia a seconda se il valore logico sia 0 o 1:

Fig. 2.3: RS485 Bifilare

Le impedenze di carico da 120 ohm poste agli estremi della linea hanno lo scopo di eliminare le eventuali riflessioni dovute alla propagazione dei segnali lungo i cavi consentendo quindi alla linea di trovarsi in adattamento di impedenza.

Nel caso "a riposo", cioè quando il dispositivo non sta trasmettendo, i 4 interruttori sono aperti e di conseguenza i 2 fili non sono connessi ad alcun potenziale. Solitamente è preferibile evitare di lasciare in una connessione flottante la linea; pertanto, viene applicata un tensione col + su A e – su B e per ottenere questa connessione si inseriscono 2 impedenze: una fra +5V e d il segnale A ed una fra il segnale B e la massa (GND).

Quando il dispositivo deve trasmettere un dato prende momentaneamente il controllo della linea, portando nello stato di ON (chiuso) gli interruttori a due a due in diagonale.
Se si desidera trasmettere lo 0 logico la linea deve essere configurata nella maniera seguente: A=GND, B=5V, altrimenti per trasmettere l'1 logico A=5V e B=GND. Così facendo la tensione prelevata tra A e B sarà +5V per bit=1 e -5V per bit=0.

Ogni dispositivo presenta anche un'interfaccia di ricezione in grado di misurare V_A - V_B:

- Se $V_A > V_B$ la ricezione si trova nello stato di 1 logico.
- Se $V_A < V_B$ la ricezione si trova nello stato di 0 logico.

Quello che succede sostanzialmente quindi, è che il blocco di ricezione permette di eseguire il

processo inverso, decodificando l'informazione inviata dal trasmettitore.
In questo modo andiamo a definire una connessione di tipo differenziale grazie alla quale non si ha necessariamente bisogno di collegare i riferimenti comuni GND dei vari dispositivi, in quanto assume valore predominante la tensione differenziale tra il nodo A e il nodo B.

Questa tecnica risulta particolarmente efficiente qualora il cavo venisse realizzato tramite una coppia di conduttori intrecciati (cavo twistato), perché un'eventuale interferenza elettromagnetica induce una tensione di disturbo su entrambi i fili della coppia creando una sovrapposizione di tensioni indesiderate con la stessa polarità rispetto al riferimento e garantendo quindi l'elisione del disturbo generato.

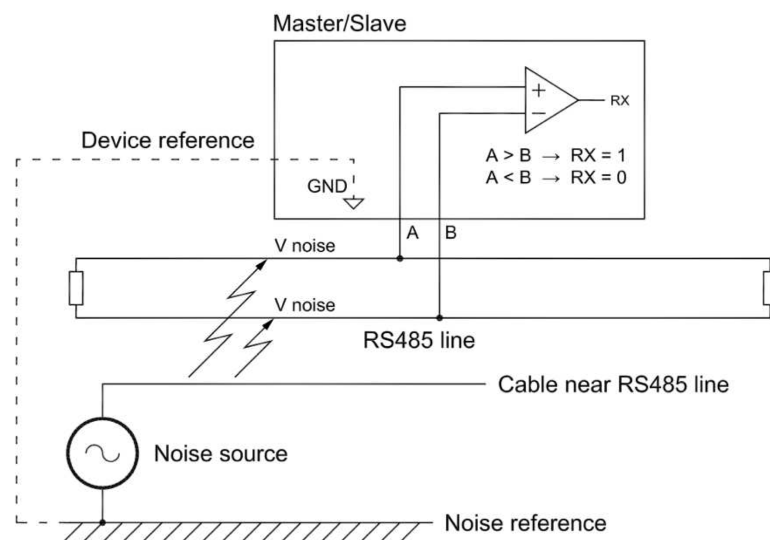

Fig. 2.4: Presenza di rumore su linea RS485

La rete seriale RS485 permette la realizzazione di impianti dove le distanze fra i dispositivi possono essere notevoli, in funzione della velocità di trasmissione dei dati (Baud Rate) e dalle relative caratteristiche dei driver hardware dei dispositivi. Nel caso del Modbus tale velocità di comunicazione varia da 9600 b/s a 115200 b/s.
Tuttavia, anche lo standard RS485 ha dei limiti in termini di dispositivi collegabili: esso prevede infatti che il trasmettitore sia in grado di pilotare fino a 32 unità di carico (UL), assumendo che 1 unità corrisponda ad un'impedenza di circa 12 kOhm.

2.2.3 Modbus DataLink Layer

Questo layer specifica come i byte vengono trasferiti da un dispositivo all'altro, indipendentemente dal loro significato: si tratta quindi di uno dei livelli software del protocollo Modbus.

È bene precisare che questo layer è a sua volta suddiviso in due "sub layer" ovvero

a) "Master/slave protocol" (protocollo master/slave)
b) "The trasmission mode: RTU or ASCII" (modalità di codifica)

a)

Come già detto, nel Modbus in comunicazione seriale abbiamo un solo master connesso alla volta e un numero di slave che in questo caso può variare da 1 a 247.

Il nodo master invia una richiesta ai nodi slave in due modalità:

- *Modalità unicast*: il master invia una richiesta ad un solo slave, specificandone l'indirizzo (vedi frame message) e questo risponde fornendo ciò che il master richiede oppure fornisce un messaggio di errore.

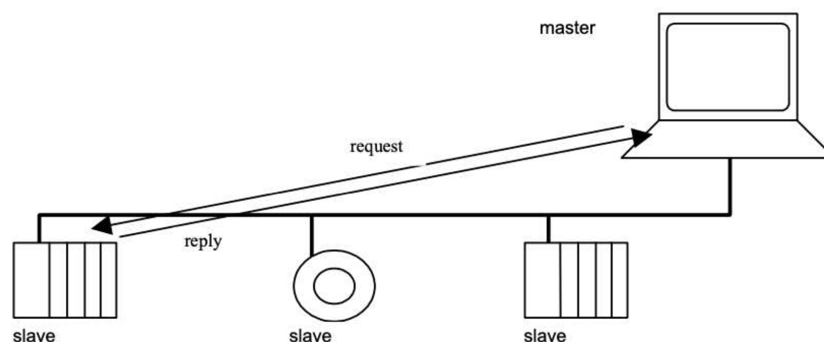

Fig. 2.5: Comunicazione Master-Slave Unicast

- *Modalità broadcast*: il master invia una richiesta a tutti gli slave, specificando nel byte dedicato all'indirizzo il valore "0". In questo caso però non può essere fornita alcuna risposta da parte degli slave infatti può trattarsi solo di un comando di scrittura (writing request), che gli slave sono obbligati ad eseguire.

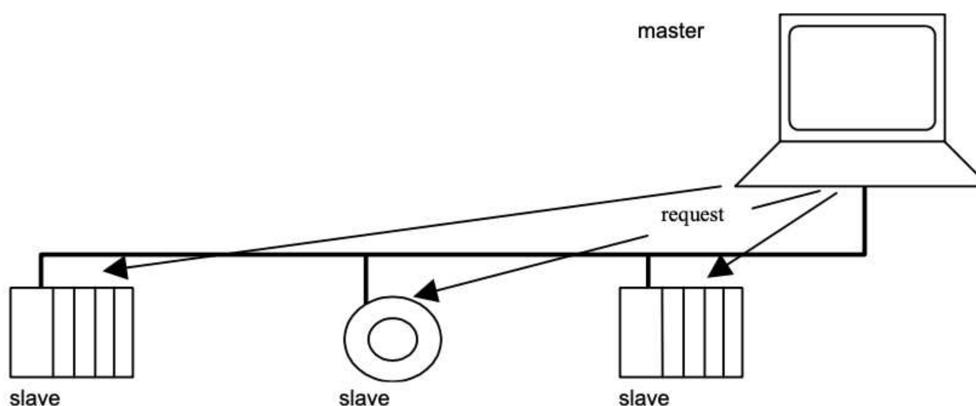

Fig. 2.6: Comunicazione Master-Slave Broadcast

Per poter andare a specificare il tipo di richiesta, viene dedicato un apposito byte nel frame message definito come "codice funzione". Nel paragrafo 2.3 verrà analizzata questa specifica.

b)

Nel secondo sub layer viene invece definito il contenuto dei campi del messaggio e determina come le informazioni sono racchiuse all'interno di essi, cioè secondo quale codifica.

È bene precisare che la modalità di trasmissione e i parametri della porta seriale devono essere gli stessi per tutti dispositivi collegati alla linea seriale.

Nonostante la modalità ASCII sia richiesta da alcune specifiche applicazioni, è richiesto che tutti i dispositivi implementino comunque la modalità RTU (possiamo quindi considerare l'ASCII come se fosse un optional).

RTU mode

Nella modalità RTU i dati vengono inviati in formato binario con un'elevata densità di caratteri, permettendo una maggiore efficienza rispetto alla modalità ASCII a parità di velocità.

Ogni messaggio è costituito da 11 bit così suddivisi:

- 1 bit di START
- 8 bit per i DATI, inviati secondo specifica Little Endian
- 1 bit di PARITA', dove solitamente si sceglie *No parity mode* ma sono selezionabili anche parità pari o parità dispari
- 1 bit di STOP

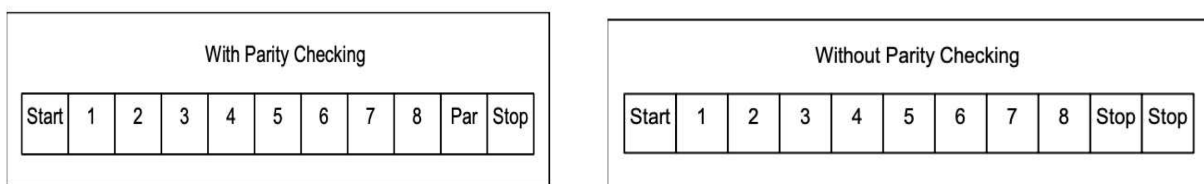

Fig. 2.7: Frame Message Modbus RTU (a sx con bit parità, a dx senza)

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low CRC Hi

Fig. 2.8: Frame message Modbus RTU

Il dispositivo che trasmette il messaggio colloca quest'ultimo in un frame che ha un punto di inizio e un punto di fine, di modo che il dispositivo che riceve un nuovo frame sia in grado di comprenderne l'inizio e la fine. Se sono presenti dei messaggi parziali, questi devono essere riconosciuti e segnalati come errori.

In questa modalità i frame del messaggio sono separati da un intervallo di tempo (*silent interval*) pari ad almeno 3.5 volte il tempo di trasmissione di un carattere.

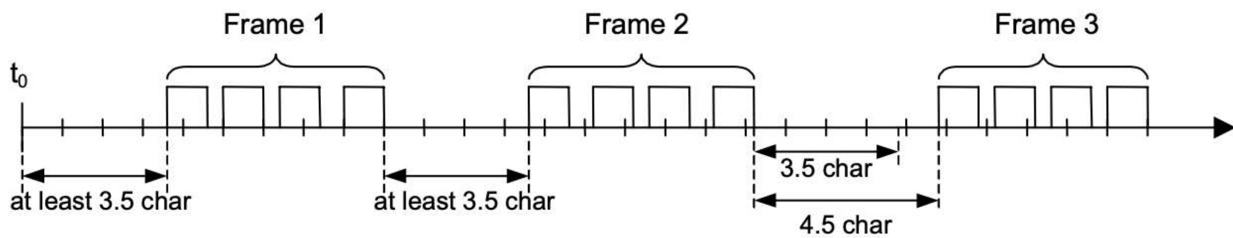

Fig. 2.9: Silent Interval

L'intero frame del messaggio deve essere trasmesso come un flusso continuo di caratteri. Se si verifica un intervallo di silenzio superiore a 1.5 volte il tempo di trasmissione di un carattere tra due caratteri, il messaggio è dichiarato incompleto e pertanto dovrà essere scartato dal ricevitore.

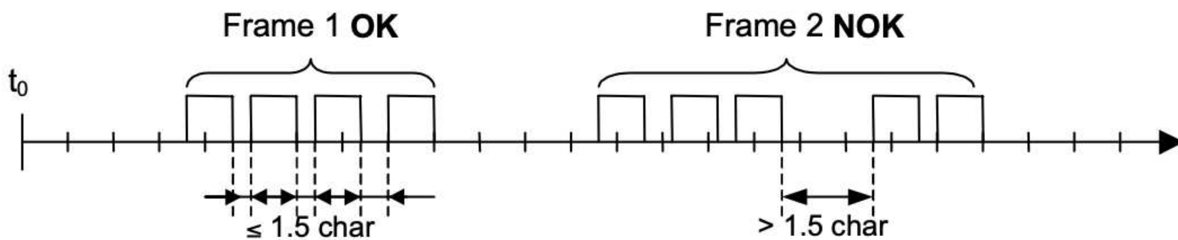

Fig. 2.10

La modalità RTU include un campo di controllo degli errori basato sul CRC, *Cyclic Redundancy Check,* il quale controlla il contenuto dell'intero messaggio e viene applicato indipendentemente dal tipo di parità scelta.

Il valore del CRC viene calcolato dal trasmettitore che lo aggiunge al messaggio; il ricevitore calcola poi un CRC durante la ricezione e confronta il proprio valore con che arriva dal trasmettitore. Se i due valori sono diversi si verifica un errore.

Essenzialmente il metodo di calcolo è basato su operazioni polinomiali: i dati vengono considerati come un polinomio ed il CRC viene calcolato dividendo tale polinomio per un polinomio "generatore" predefinito, tenendo il resto come valore di controllo.

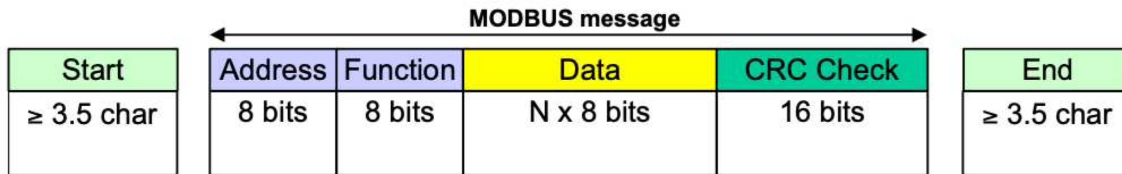

Fig. 2.11: Messaggio Completo Modbus RTU

ASCII mode

Quando viene usata questa modalità, ogni byte all'interno di un messaggio viene inviato come due caratteri ASCII. Pertanto questa modalità risulta essere meno efficiente rispetto al Modbus RTU e trova applicazione qualora non vi sia la possibilità di conformarsi ai requisiti dei timer in modalità RTU.

Ogni messaggio è costituito da 10 bit così suddivisi:

- 1 bit di START
- 7 bit per i dati, inviati secondo specifica LittleEndian
- 1 di parità
- 1 bit di STOP

La struttura del messaggio è molto simile alla modalità in RTU, si faccia quindi riferimento alla fig. 2.7

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Fig. 2.12: Frame message Modbus ASCII

Ogni messaggio in modalità ASCII infatti inizia con " : " e termina con una coppia di caratteri "carriage return- line feed": nel momento in cui viene ricevuto " : " ciascun dispositivo decodifica il carattere successivo fino a quando non rileva la fine del frame.

I caratteri consentiti per tutti gli altri campi sono quelli esadecimali codificati in ASCII.

Fra un messaggio e l'altro è previsto di default un silent interval di 1 secondo, altrimenti se superato viene segnalato un errore (è possibile modificare quest'impostazione).

Il principio di funzionamento per il controllo degli errori è sostanzialmente lo stesso esposto per la modalità RTU, con la differenza che non è più un codice a ridondanza ciclica bensì ridondanza longitudinale: LRC.

Esso viene calcolato sommando i successivi byte del messaggio, scartando eventuali riporti, e prendendo il complemento a 2 del risultato. Questo calcolo viene effettuato sul byte del messaggio

prima della codifica di ciascun byte nei corrispondenti caratteri ASCII. (nel calcolo non sono inclusi i caratteri di inizio e fine del frame, rispettivamente " : " e " CR-LF ".)

2.2.4 Modbus Application Layer

In questo livello viene definito come i dati vengono strutturati e scambiati tra i dispositivi in rete, stabilendo le regole e i formati per la comunicazione per rendere il più possibile efficiente la comunicazione. Questo livello si occupa sostanzialmente della PDU (Protocol Data Unit).

La PDU contiene le informazioni utili al Master affinché esso possa effettuare correttamente le richieste agli Slave ovvero 1 byte dedicato all'operazione da svolgere ed un campo dati specifico per il comando da eseguire.

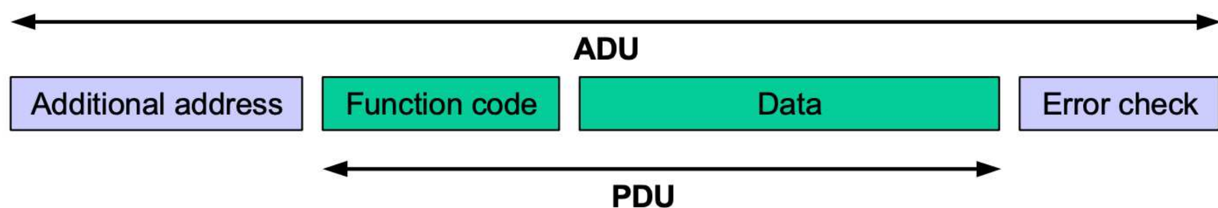

Fig. 2.13: Rappresentazione Application Layer

Come si vede nell'immagine soprastante, viene definita anche la ADU: Application Data Unit.
Il campo funzione contiene un valore di default per il comando che si va ad implementare. Il Modbus prevede che i valori da 1 a 127 siano usati per la codifica del comando e che da 128 a 255 siano dedicati al segnalamento di eventuali errori rilevati durante l'esecuzione del programma.

I comandi standard del protocollo Modbus relativi alla lettura e scrittura dei valori riguardanti gli Slave si basano sul concetto di "oggetto" e non all'indirizzo di memoria. In questo modo viene separato l'indirizzamento delle risorse dalla loro posizione fisica nella memoria dello slave.
Dunque, ogni slave deve definire gli oggetti, ai quali il Master "può accedere", tramite l'uso del protocollo assegnandogli un indirizzo Modbus e permettendo la gestione della corrispondenza di tali oggetti e la loro memoria.
Gli oggetti sono raggruppati nella seguente tabella, spesso chiamata anche come "memory map":

Primary tables	Object type	Type of	Comments
Discretes Input	Single bit	Read-Only	This type of data can be provided by an I/O system.
Coils	Single bit	Read-Write	This type of data can be alterable by an application program.
Input Registers	16-bit word	Read-Only	This type of data can be provided by an I/O system
Holding Registers	16-bit word	Read-Write	This type of data can be alterable by an application program.

2.3 Funzioni

Per definire il tipo di azione che gli slave devono compiere su richiesta del Master si usano dei codici funzione. Di seguito vengono riportate le funzioni che verranno poi implementate nella simulazione tramite uVision 5.

- *Funzione 01: Read coils (Lettura Uscite Digitali)*

Questa funzione serve a leggere da 1 a 2000 stati contigui di "bobine" in un dispositivo remoto. Il PDU di richiesta indica l'indirizzo di partenza, ovvero l'indirizzo della prima bobina da leggere, e il numero di bobine richieste. Nel PDU, le bobine sono numerate a partire da zero, quindi le bobine 1-16 sono indirizzate come 0-15.Gli indirizzi di memoria vanno da 1 a 10000.

Request

Function code	1 Byte	**0x01**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of coils	2 Bytes	1 to 2000 (0x7D0)

Response

Function code	1 Byte	**0x01**
Byte count	1 Byte	**N***
Coil Status	**n** Byte	n = N or N+1

Fig. 2.14: Fc01

Nella risposta, gli stati delle bobine sono disposti con un bit per ogni bobina nel campo dati. Lo stato di ogni bobina è rappresentato da 1 = ON e 0 = OFF. Il bit meno significativo (LSB) del primo byte contiene l'output della bobina richiesta nella query.

Nel caso in cui si verifichi un errore o un'eccezione la risposta è costituita nel modo seguente:

Function code	1 Byte	**Function code + 0x80**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.15

- *Funzioni 02: Read Discrete Input (Lettura Ingresso Digitale)*

La struttura della richiesta e della risposta è analoga alla funzione 01, con la differenza che si sta parlando di ingressi e non di uscite e pertanto si tratta di "oggetti di memoria" di sola lettura.
Gli indirizzi di memoria vanno da 10001 a 20000.
In caso di errore o eccezione la risposta è:

Error code	1 Byte	**0x82**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.16

- *Funzione 03: Read Holding Registers (Lettura Uscita Analogica)*

Questo codice funzione viene utilizzato per leggere il contenuto di un blocco contiguo di registri holding in un dispositivo remoto. Il PDU di richiesta specifica l'indirizzo del registro di partenza e il numero di registri.

Request

Function code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x **N***
Register value	**N*** x 2 Bytes	

***N** = Quantity of Registers

Fig. 2.17: Fc03

I dati dei registri nel messaggio di risposta sono impacchettati come due byte per ogni registro, con il contenuto binario allineato a destra all'interno di ciascun byte. Per ogni registro, il primo byte contiene i bit di ordine alto e il secondo contiene i bit di ordine basso.

Gli indirizzi di memoria vanno da 40001 a 50000.

Nel caso in cui si verifichi un errore oppure un'eccezione:

Error code	1 Byte	**0x83**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.18

- *Funzione 04: Read Input Registers (Lettura Ingresso Analogico)*

Il codice della funzione serve a leggere una sequenza di registri di input contigui da un dispositivo remoto, con un massimo di 125 registri alla volta. Nella richiesta, vengono indicati l'indirizzo iniziale del registro e quanti registri leggere.

La risposta del dispositivo contiene i dati dei registri, impacchettati in due byte per ciascun registro. Il primo byte di ogni registro contiene i bit più significativi e il secondo byte quelli meno significativi.

Gli indirizzi di memoria vanno dal 30001 al 40000.

Request

Function code	1 Byte	**0x04**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	0x0001 to 0x007D

Response

Function code	1 Byte	**0x04**
Byte count	1 Byte	2 x **N***
Input Registers	**N*** x 2 Bytes	

***N** = Quantity of Input Registers

Error

Error code	1 Byte	**0x84**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.19: Fc04

- *Funzione 05: Write Single Coil (Scrittura Uscita Digitale)*

Questa funzione serve a cambiare lo stato di un'uscita specifica di un dispositivo remoto, impostandola su acceso, ON, o spento, OFF. Il valore che indica lo stato richiesto è inserito nel campo dati della richiesta. Se il valore è *FF 00*, l'uscita sarà accesa; se è *00 00*, l'uscita sarà spenta. Qualsiasi altro valore è considerato non valido e non cambierà lo stato dell'uscita.

Request

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Response

Function code	1 Byte	**0x05**
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Error

Error code	1 Byte	**0x85**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.20: Fc05

- *Funzione 06: Write Single Register (Scrittura Uscita Analogica)*

Questa funzione è usata per scrivere un'uscita analogica di un dispositivo slave remoto.

Nella PDU della richiesta inviata dal master viene specificato l'indirizzo del registro che deve essere scritto, partendo dal registro zero. La risposta, se non si verificano errori, è un echo della richiesta cioè non appena i dati del registro sono stati aggiornati, viene ritornata una risposta uguale alla richiesta.

Request

Function code	1 Byte	**0x06**
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Response

Function code	1 Byte	**0x06**
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Error

Error code	1 Byte	**0x86**
Exception code	1 Byte	01 or 02 or 03 or 04

Fig. 2.21: Fc06

2.4 Gestione degli errori

Come illustrato nel precedente paragrafo, il master invia un sequenza di codici che corrispondono ad una specifica richiesta e che lo slave deve (o meglio dovrebbe eseguire), ed una volta conlcusa tale operazione, invia una risposta al master stesso. Tuttavia non sempre le operazioni "vanno a buon fine" e come visto può essere generato un codice di errore o delle eccezioni.
Dunque le dinamiche a cui si va incontro in una comunicazione master/slave sono le seguenti:

1) Lo slave *riceve* la richiesta *senza errori* di comunicazione ed è in grado di gestirla. Una volta eseguita l'operazione restituisce appunto una risposta "normale".
2) Lo slave *non riceve* la richiesta da parte del client a causa di un errore di comunicazione e pertanto non viene generata alcuna risposta dal server. Essendo presente però una condizione di tempo limite in cui deve avvenire la comunicazione fra master e slave, non appena questo tempo viene superato, verrà eseguita la funzione annessa alla condizione di timeout.
3) Lo slave *riceve* la richiesta, ma *rileva un errore* di comunicazione (parità, CRC…) e quindi non restituisce alcuna risposta. Anche qui il master esegue la funzione collegata alla condizione di timeout.

4) Lo slave *riceve* la richiesta *senza alcun errore* di comunicazione, ma *non riesce a gestirla* (per esempio a causa della richiesta di lettura di un registro insesistente). Lo slave allora restituirà un'eccezione tramite la quale va ad informare il master della natura dell'errore.

Di seguito si espone come è strutturata una risposta del server contente un'eccezione.

2.4.1 Eccezione da parte dello slave

Il messaggio di eccezione è costituito da due campi:

- *Campo codice funzione*:

In una risposta normale lo slave ripete il codice funzione della richiesta, inviata dal master, nel campo del codice funzione della risposta. In queste condizioni, tutti i codici funzione hanno il MSB pari a 0 e i loro valori sono tutti minori di 80 in esadecimale. In una risposta contenente un'eccezione, invece, lo slave imposta MSB a 1. Il codice funzione che viene ritornato al master presenta un "offset" pari ad 80 in esadecimale rispetto al codice funzione "normale". In questo modo il master è in grado di riconoscere la presenza di un'eccezione ed esaminare il campo dati per capire di che eccezione si tratti.

- *Campo dati*:

Come detto sopra, all'interno di questo campo viene inserito il codice per risalire alla causa dell'eccezione, come illustrato nelle tabelle delle funzioni del precedente paragrafo.

Nella tabella sottostante vengono riportati i significati dei codici d'eccezione inerenti alle funzioni che verrano poi implementate nel successivo capitolo ovvero *01,02,03,04*.

Code	Nome	Significato
01	FUNZIONE ILLEGALE	Il codice di funzione ricevuto non è permesso per lo slave.
02	INDIRIZZO DATI ILLEGALE	L'indirizzo dati specificato non è valido per il server. (es: quantità di registri superiore al limite)
03	VALORE DATI ILLEGALE	Un valore contenuto nel campo dati della richiesta non è valido per lo slave.
04	GUASTO DEL DISPOSITIVO SLAVE	Si è verificato un errore non recuperabile durante il tentativo di esecuzione della richiesta.

2.4.2 Calcolo del CRC

Nel messaggio di una comunicazione in Modbus RTU è presente un campo di controllo errori basato su una tecnica di controllo chiamata CRC: Cyclic Redundancy Check ("controllo a ridondanza ciclica").
Questo campo verifica il contenuto dell'intero messaggio, indipendentemente dal tipo di parità (pari, dispari o nessuna) ed è costituito da 16 bit implementati da 2 byte.

Il CRC viene aggiunto al messaggio come ultimo campo, mettendo prima il byte meno significativo seguito da quello meno significativo (come avviene nella codifica Little Endian).
Il valore contenuto all'interno di questi due byte viene calcolato dal dispositivo mittente, il quale appunto aggiunge il campo alla fine del messaggio. Il dispositivo ricevente ricalcola il CRC, durante la ricezione stessa del messaggio, e se i due CRC sono diversi si verifica un errore.

Il calcolo di questo campo inizia precaricando tutti e 16 i bit ad '1'e successivamente viene calcolato il CRC solo sugli 8 bit di dati del messaggio, quindi escludendo bit di start, stop e parità.

Tale calcolo viene eseguito nel modo seguente:

- Ogni carattere da 8 bit viene sottoposto ad un'operazione di XOR con il contenuto del registro
- Successivamente, il risultato viene shiftato nella direzione del bit meno significativo aggiungendo uno zero nella posizione del MSB.
- Viene prelevato LSB: se il valore è pari a 1, tale LSB viene messo in XOR con un valore fisso preimpostato. Se il valore è 0 non avviene nessun' operazione.
- Questo processo si ripete per 8 volte e successivamente si mette in OR esclusivo il byte successivo con il valore aggiornato del registro.
- Si ripete nuovamente il procedimento anche per il secondo byte.
- Il contenuto finale del registro è il valore del CRC.

Nel capitolo successivo verrà riportato un estratto di codice riguardante quanto appena esposto.

Capitolo 3

Implementazione Modbus RTU su RS485 con STM32F103 e STM32F334

3.1 Interfacciamento STM32 con RS485

Nella prima parte di questo capitolo viene esposto come è stato configurato il microcontrollore STM32f103 e come implementare il protocollo in questione, andando ad inviare un messaggio sulla USART simulata del micro.
Nella seconda parte viene invece configurato l'STM32f334 andando a mostrare le differenze rispetto al caso precedente.

A scopo illustrativo, si fa presente che per realizzare fisicamente una comunicazione in Modbus tra il micro e degli slave su RS485, c'è bisogno di un convertitore da TTL a RS485 che implementa un chip chiamato MAX485 ed è riportato nella figura sottostante.

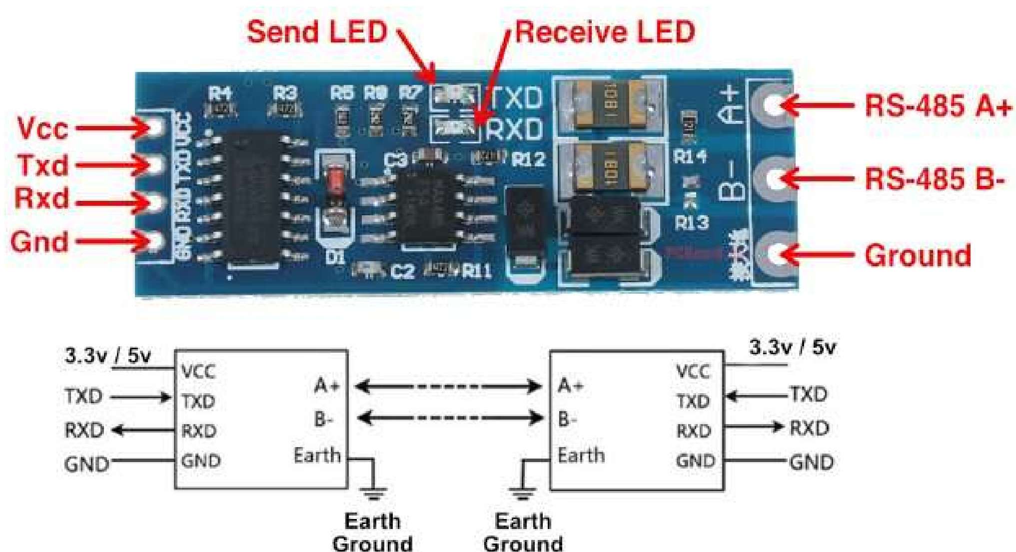

Fig. 3.1: Convertitore TTL-RS485

3.2 Configurazione STM32F103

Tramite il software messo a disposizione dalla STM, STM32CubeMX, si ha la possibilità di configurare il microcontrollore a seconda dell'applicazione di cui si avrà bisogno.

In questo caso si è scelto STM32f103c8t7, dove le ultime quattro "cifre" indicano:

- C8: indica la memoria flash disponibile nel micro, ovvero 64KB.
- T: indica il tipo di package, ovvero Low-profile Quad Flat Package con 48 pin.
- 7: specifica il range di temperatura operativa, ovvero [-40, +85] °C

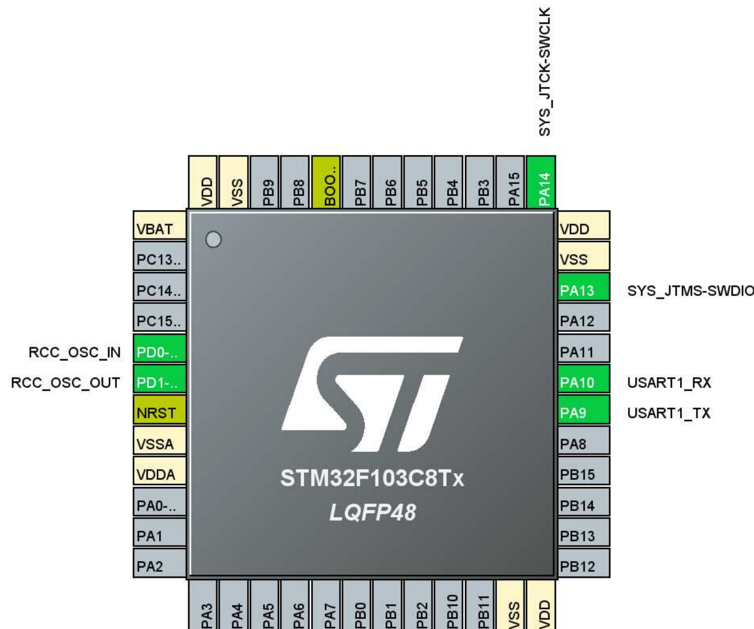

Fig. 3.2: Configurazione F103

Per fare in modo che il microcontrollore trasmetta i dati tramite lo standard RS485 c'è bisogno di configurare una periferica apposita a questo scopo. Si configura quindi la USART, scegliendo ad esempio la USART1, in modalità asincrona e scegliendo 9600 come velocità di trasmissione dati, 8 bit per il frame dati, nessun bit di parità e 1 bit di stop.

Si sceglie poi di instaurare una comunicazione di tipo half-duplex dove la USART1 del micro gestisce sia la trasmissione che la ricezione dei dati sulla stessa linea, ma in un verso alla volta. È necessario quindi controllare un segnale di abilitazione del trasmettitore per passare da trasmissione a ricezione e viceversa. I transceiver RS485 come MAX485 hanno un pin dedicato alla trasmissione (Driver Enable) e uno per la ricezione (Receiver Enable).

A tale scopo il microcontrollore deve abilitare il transceiver per la trasmissione e disabilitarlo per la ricezione e per farlo bisogna configurare la USART1 per controllare automaticamente tale pin: ad esempio tramite interruzioni o come in questo caso con l'uso di un GPIO tramite i pin PA9 e PA10.

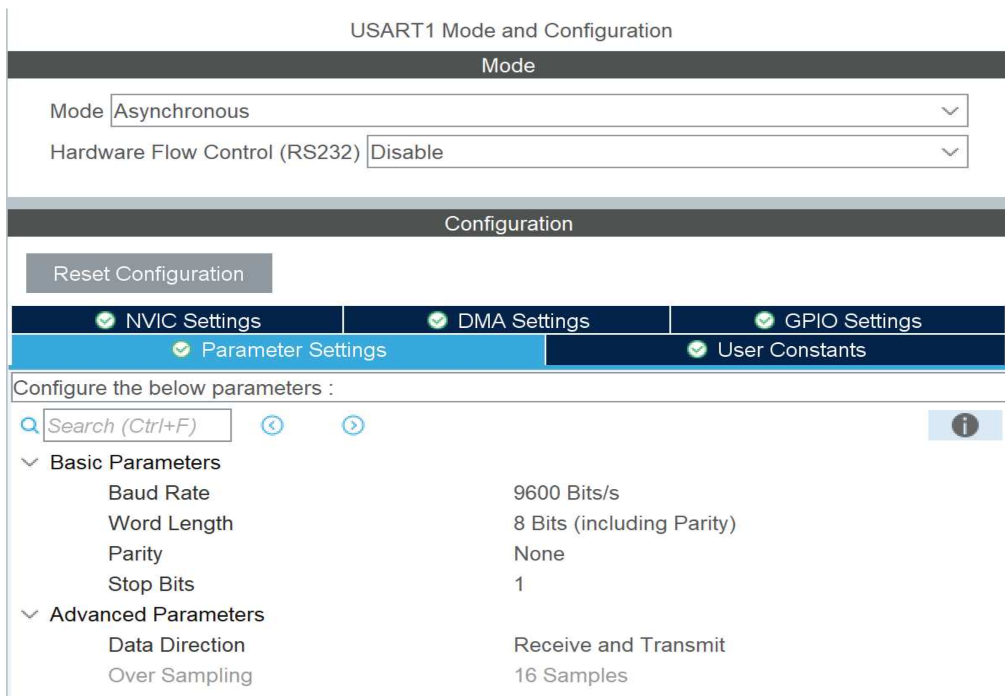

Fig. 3.3: Configurazione USART1

Andiamo a settare anche il DMA (Direct Memory Access), impostando la massima priorità sia per il ricevitore che per il trasmettitore.

Come visibile in fig. 3.3 il software che si sta usando non da la possibilità di selezionare il DE per RS485 quindi verrà creata un'apposita funzione, a tale scopo, esposta nei paragrafi successivi.

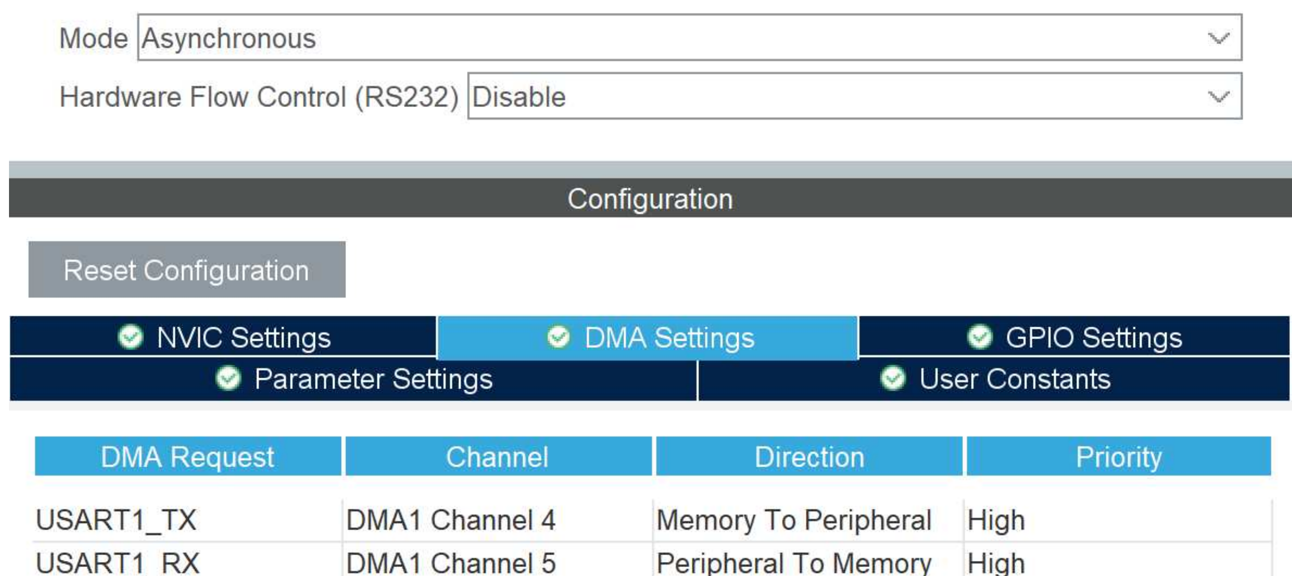

Fig. 3.4: Configurazione DMA

È necessario configurare almeno 2 timer di modo da controllare che vengano rispettati i tempi previsti dal protocollo Modbus: tempo fra un carattere e l'altro dello stesso frame ($t_{1,5}$) e fra 2 frame diversi ($t_{3,5}$). Dovendo solamente inviare un messaggio sulla USART simulata, non si avrà modo di verificare i tempi di risposta ecc… e quindi viene configurato un solo timer (TIM3) a scopo puramente didattico che controlli il tempo che intercorre tra un frame e l'altro.

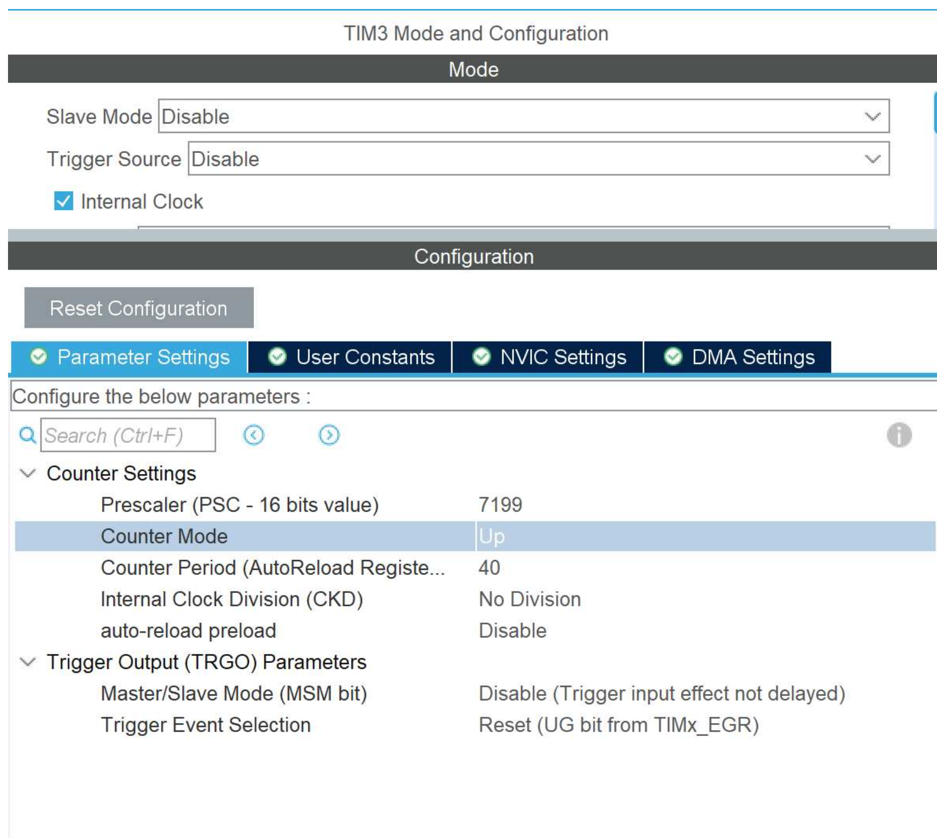

Fig. 3.5: Configurazione TIM3

Il prescaler del TIM3, che lavora a 72MHz, viene settato a 7199 in modo da avere un incremento del contatore interno ogni 0,1 ms (cioè con una frequenza di 10 kHz).

La formula utilizzata per il calcolo è la seguente:

$$\text{Prescaler} = \frac{f_{\text{timer}}}{f_{\text{tick}}} - 1$$

Dove f_{tick} è appunto pari a 10 kHz.

Nell' ARR (AutoReloadRegister) viene inserito il valore 40, di modo da generare un interrupt dopo 4,01 ms ovvero il tempo limite che può intercorrere tra un messaggio e l'altro (= $t_{3,5}$).

Per ottenere tale valore si è effettuato il seguente calcolo:

$$T_{\text{char}} = \frac{\text{Numero di bit per carattere}}{\text{Baud Rate}} = \frac{11}{9600}$$

$$\text{ARR} = T_{3,5\text{char}} \times f_{\text{tick}} = 4,01 \text{ ms} \times 1\,000\,000 \text{ tick/secondo} = 4010$$

Nella sezione Reset and Clock Control si sceglie Crystal Ceramic Resonator di modo da fornire unclock stabile e preciso al microcontrollore.

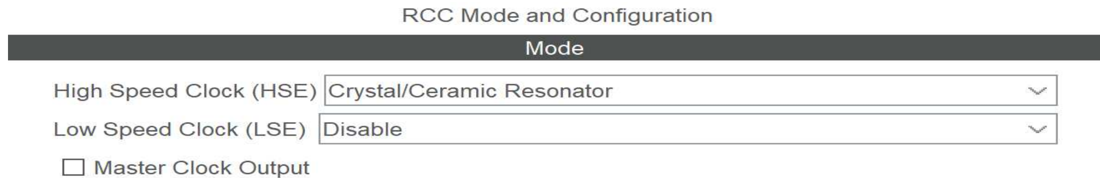

Fig. 3.6: RCC Mode Configuration

Spostandosi nella finestra relativa alla configurazione del clock si imposta nel PLL source l'HSE e si fa in modo che il micro lavori a 72 MHz, andando a selezionare un moltiplicatore pari a 9.

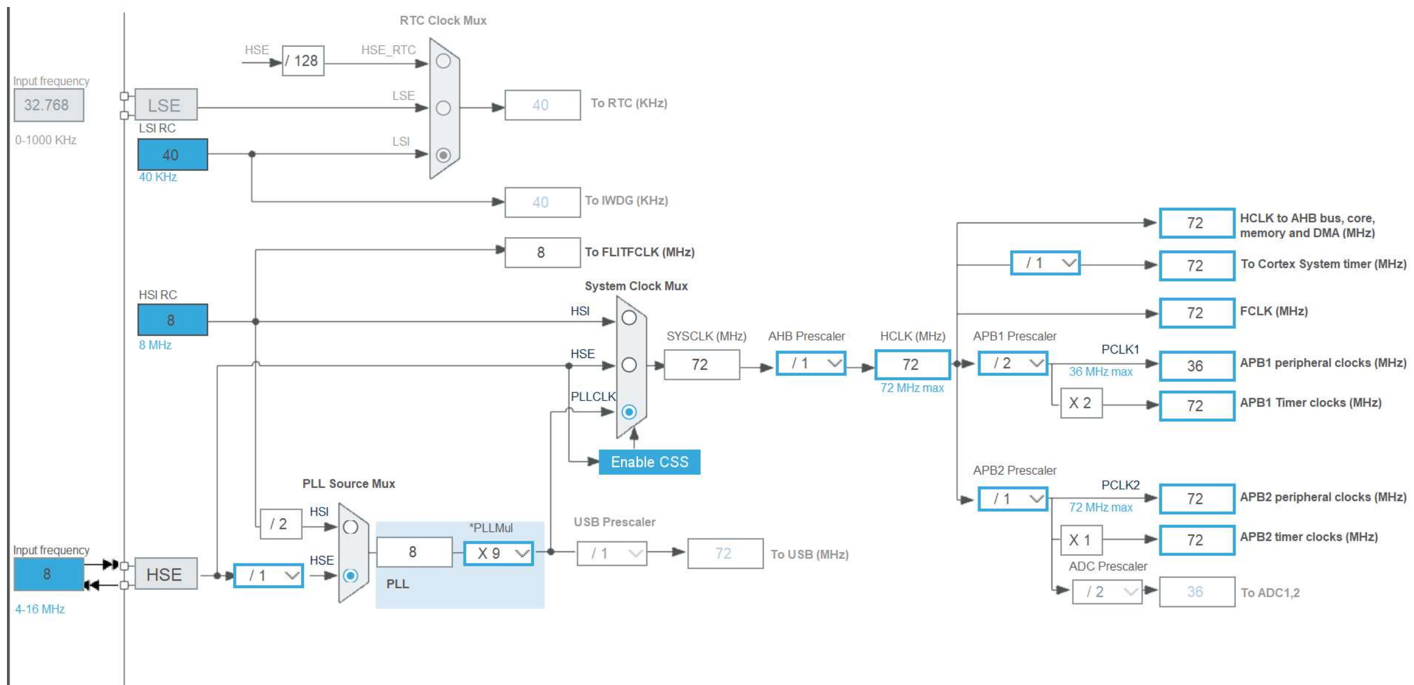

Fig. 3.7: Configurazione Clock F103

A questo punto si può procedere alla generazione del codice (MDK-ARM) da andare ad implementare in uVision 5.

3.3 Creazione del progetto in uVision 5

Una volta creato il progetto all'interno di uVision, è necessario implementare una libreria Modbus RTU. In rete ci sono versioni sia a pagamento che gratuite e per quest'applicazione è stata scelta *FreeModbus* facilmente scaricabile da GitHub.
Essendo una libreria open source, ogni sviluppatore va poi a "metterci le mani" per adattarla al caso proprio. Dunque, si effettuano le apposite modifiche per la famiglia di microcontrollori di nostro interesse ovvero la famiglia F103XX.

Dalla cartella zip contenente *FreeModbus*, si copia la cartella "Middlewares" nella directory principale del progetto che si è creato da STM32CubeMX.

Si devono ora importare i file sorgente e i file di intestazione all'interno del progetto di uVision:

- i file sorgente (.c) si importano andando su *Manage Project Items*. Si aggiunge un nuovo gruppo, per comodità chiamato "Modbus" e vi si vanno ad implementare i file .c

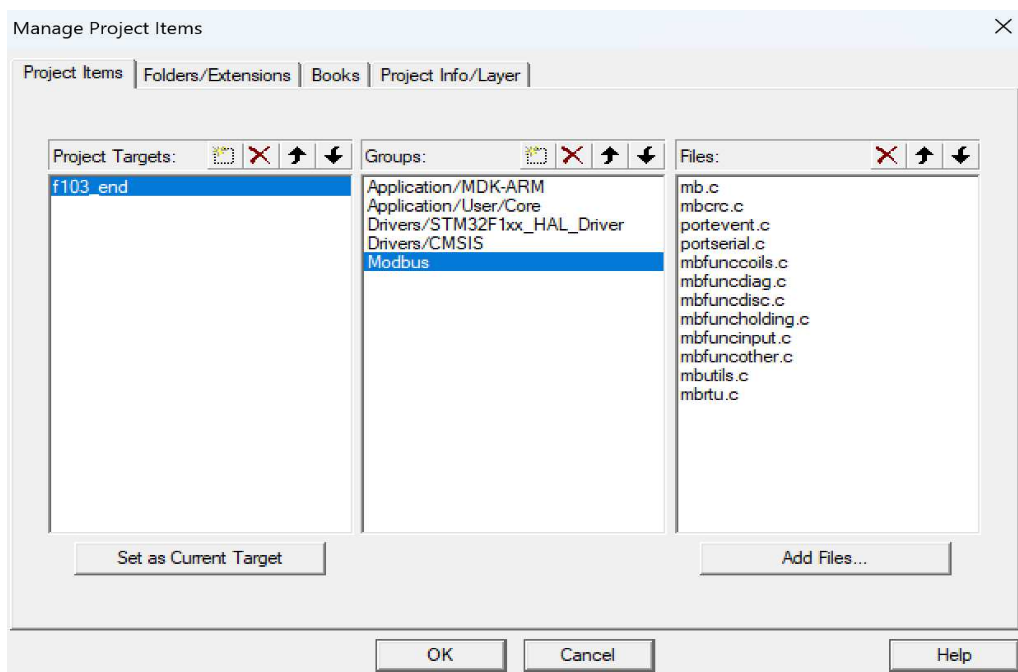

Fig. 3.8: Implementazione file sorgente

- per i file di intestazione (.h) invece è necessario passare il percorso tramite cui il software andrà a prendere questi file. Ci si sposta sulla finestra *Options for Target* e nella sezione *C/C++*. Da qui su *Include Paths* si aggiungono i percorsi per arrivare alle directory in cui sono contenuti i file di nostro interesse.

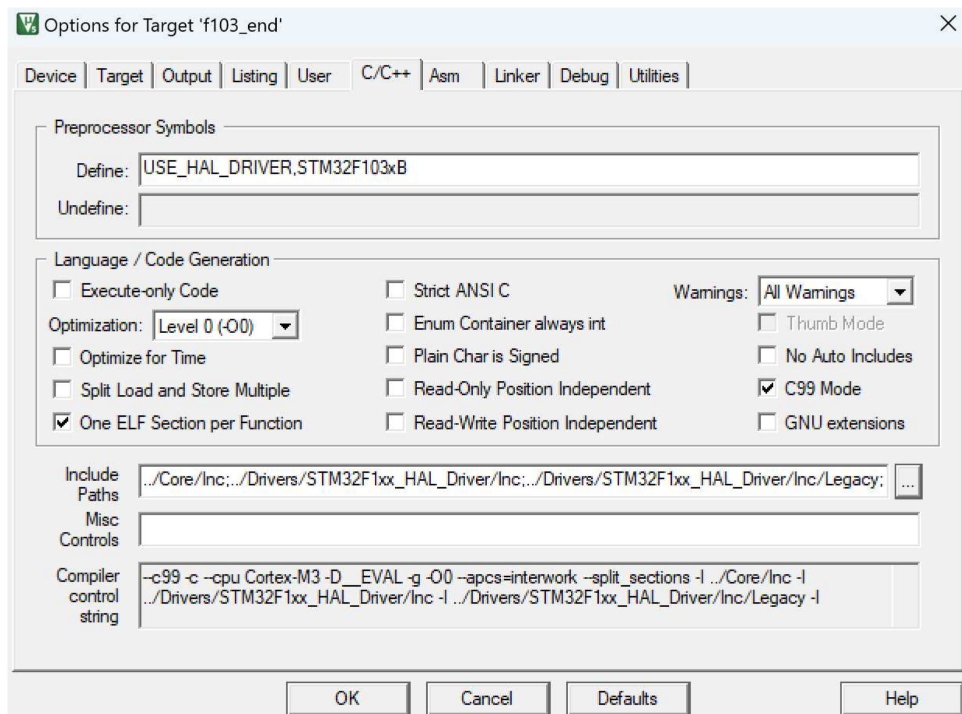

Fig. 3.9: Implementazione file intestazione

Una volta importata la libreria, e adattati i file alla famiglia f103, si può procedere con la scrittura del codice all'interno del file *main.c* presente nel target *Application User/Core.*

Dopo aver incluso i file di intestazione, si dichiara la handle della USART1 ovvero quella struttura dati che consente di configurare e gestire la USART1 tramite le funzioni della libreria HAL (Hardware Abstract Layer).

Per simulare l'invio dei dati tramite la periferica USART1 è opportuno eseguire un retarget della funzione printf, che andrebbe a scrivere sullo standard output (in questo caso costituito dal *Debug Printf Viewer*), verso la USART stessa.

```
59  /* Private user code ---------------------------------------------------------*/
60  /* USER CODE BEGIN 0 */
61  // Dichiarazione dell'handle della UART, creata da STM32CubeMX
62  extern UART_HandleTypeDef huart1;
63
64  //Funzione di retargeting per inviare un carattere tramite UART
65  int __io_putchar(int ch)
66  {
67      HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
68      return ch;
69  }
70
71  // Funzione standard C che usa il retargeting per printf
72  int fputc(int ch, FILE *f)
73  {
74      return __io_putchar(ch);
75  }
```

Fig. 3.10

Si vanno poi a definire i buffer per il trasmettitore e il ricevitore e l'array che conterrà il messaggio da inviare allo slave.

```
78  uint8_t RxData[32];
79  uint8_t TxData[8];
80  uint16_t Data[10];
81
82  uint16_t crc16(const uint8_t* data, uint16_t length);
83
84  // l'algoritmo seguente esamina byte per byte i dati eseguendo operazioni di XOR e shift sui bit
85
86  uint16_t crc16(const uint8_t* data, uint16_t length) {
87      uint16_t crc = 0xFFFF; // CRC inizializzato a -1 (per modbusRTU di default si mette questo valore)
88      for (uint16_t i = 0; i < length; i++) {
89          crc ^= data[i]; // xor del byte attuale con il valore CRC
90          for (uint8_t j = 0; j < 8; j++) {
91              if (crc & 0x0001) {
92                  crc = (crc >> 1) ^ 0xA001;  // polinomio per CRC-16
93              } else {
94                  crc >>= 1;
95              }
96          }
97      }
98      return crc;
99  }
```

Fig. 3.11

Come visibile in fig. 3.11 viene poi implementato un algoritmo che vada ed eseguire i calcoli citati nel capitolo precedente per ricavare il valore del CRC.

Inoltre viene predisposta anche una funzione di *CallBackEvent* cioè, una volta che lo slave ha ricevuto il messaggio correttamente invia una conferma di avvenuta ricezione al master (anche in questo caso si tratta di qualcosa a scopo illustrativo, in quanto non ci sarà nessuno slave).

```
102 void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size){
103
104     // i primi 3 byte RICEVUTI sono dedicati a: salve ID,codice funzione, nr bytes.
105     // andiamo a fare uno shift di 8 posizioni sa sinistra e sommiamo il successivo dato al precedente
106     // in modo da ottenere un array a 16 bit
107
108 /*da commentare nel caso in cui si vadano a leggere uscite digitali...
109     si potrebbe far in modo di suddividere i byte di ritorno nei bit così da rendere più leggibile il codice
110     essendo delle uscite digitali, tuttavia non avendo uno slave che risponde questo risulta poco funzionale */
111
112     Data[0] = RxData[3]<<8 | RxData[4];
113     Data[1] = RxData[5]<<8 | RxData[6];
114     Data[2] = RxData[7]<<8 | RxData[8];
115     Data[3] = RxData[9]<<8 | RxData[10];
116     Data[4] = RxData[11]<<8 | RxData[12];
117 }
118
```

Fig. 3.12

Si aggiunge la funzione che verrà utilizzata per inviare i messaggi allo slave denominata *sendData.*

All'interno di questa funzione si abilita il driver per RS485 settando al livello logico alto GPIOA.

Successivamente si inviano i dati tramite la UART e si reimposta al livello logico basso il GPIOA di modo da abilitare la ricezione.

```
// su stm32cubemx non abbiamo la possibilità di selezionare il DE per RS485 ma solo per RS232.
// implementiamo quindi nella seguente funzione anche l'abilitazione alla trasmissione/ricezione
void sendData (uint8_t *data)
{
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET); // driver enable, DE=1: abilito la trasmissione
  HAL_UART_Transmit(&huart1, data, 8, 1000);
  HAL_GPIO_WritePin(GPIOA,GPIO_PIN_8 , GPIO_PIN_RESET);// DE=0: abilito la ricezione
}
```

Fig. 3.13

Proseguendo all'interno del ciclo *main* si inizializzano tutte le periferiche precedentemente configurate tramite STM32CubeMX.

Tramite apposita riga di codice si va a settare il TIM3 in modalità interrupt per poi preparare il buffer TxData con il messaggio da inviare allo slave.

A seconda della funzione da implementare il contenuto del messaggio sarà differente:

```
157    /* Initialize all configured peripherals */
158    MX_GPIO_Init();
159    MX_DMA_Init();
160    MX_USART1_UART_Init();
161    MX_TIM3_Init();
162    /* USER CODE BEGIN 2 */
163
164  // Avvia TIM3 in modalità interrupt
165    HAL_TIM_Base_Start_IT(&htim3);
166
167  // andiamo a ricevere i dati tramite "interrupt idle method"
168    HAL_UARTEx_ReceiveToIdle_IT(&huart1, RxData, 32);
169
170  // frame message in modbus RTU è costituito da:
171  // indirizzo: 1 byte + funzione: 1 byte + dati: fino a 252 bytes + crc: 2 byte
172
173    TxData[0] = 0x05;  // indirizzo dello slave
174
175    TxData[1] = 0x03;  // codice funzione: lettura uscita analogica(read holding registers)
176
177
178    TxData[2] = 0x00;
179
180    TxData[3] = 0x04;
181  //l'indirizzo del registro è: 00000000 00000100 = 4 + 40001 = 40005
182
183    TxData[4] = 0x00;
184    TxData[5] = 0x05;
185  //il numero dei registri da leggere è: 00000000 00000101 = 5 Registri = 10 Bytes
```

Fig. 3.14

Come indicato dall'immagine soprastante, la richiesta del Master consiste nell'andare a leggere l'uscita analogica (funzione 03) impostando come indirizzo iniziale del registro il numero 4 (con un offset di 40001 l'indirizzo finale è 40005) dello slave il cui ID è 05. Inoltre tramite TxData[4] e TxData[5] il Master va a richiedere che vengano letti un totale di 5 registri equivalenti a 10 byte dati.

A questo punto mancano gli ultimi 2 byte dedicati al CRC che sono calcolato tramite i seguenti comandi:

```
187   uint16_t crc = crc16(TxData, 6);
188   TxData[6] = crc&0xFF;    // CRC LOW
189   TxData[7] = (crc>>8)&0xFF;  // CRC HIGH
190
191 // Trasmetti il frame tramite UART
192   HAL_UART_Transmit(&huart1,TxData, 8, HAL_MAX_DELAY);
193
```

Fig. 3.15

Infine si invia il frame dati tramite la USART1:

```
191 // Trasmetti il frame tramite UART
192   HAL_UART_Transmit(&huart1,TxData, 8, HAL_MAX_DELAY);
193
194   sendData(TxData);
```

Fig. 3.16

Avviando il Debug e mandando in Run il programma è possibile vedere la sequenza di codici inviati sulla USART1 tramite l'apposita finestra.

Prima di fare questo è fondamentale andare a settare il debug tramite il simulatore di uVision e parametrizzarlo opportunamente, come di seguito riportato.

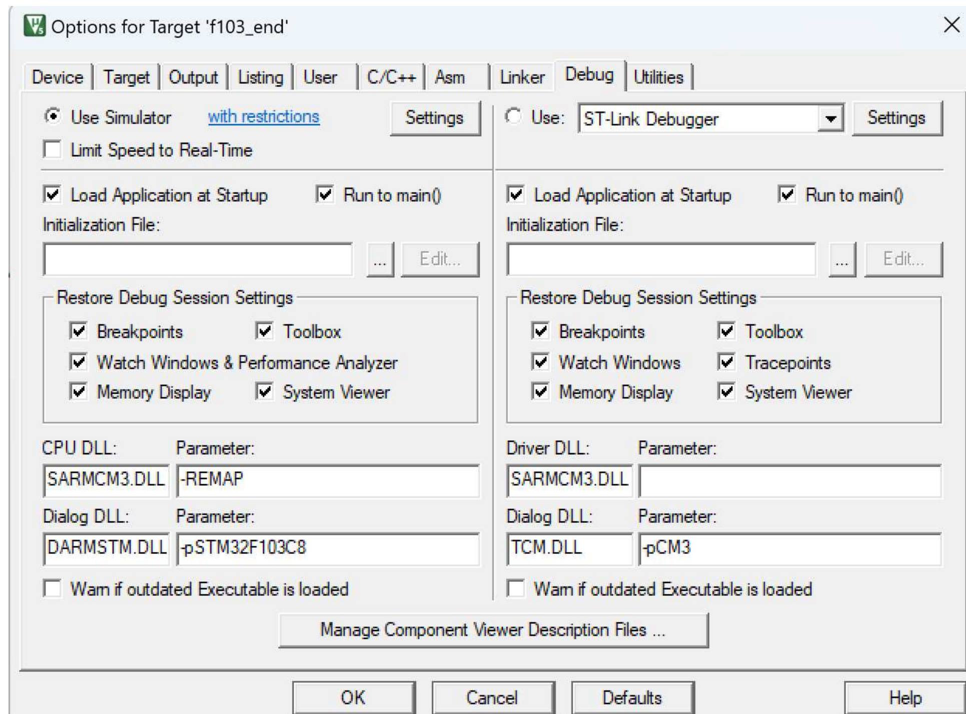

Fig. 3.17

Nella finestra del *Debug* presente all'interno di *Options for Target* si seleziona *Use Simulator* e si inseriscono le corrette stringhe per Dialog DLL e Parameter.

- Dialog DLL: è una libreria specifica che permette a uVision di interagire con l'ambiente hardware emulato per simulare un microcontrollore. Questo file DLL contiene le informazioni su come il software deve comunicare con il microcontrollore e le sue periferiche durante la simulazione o il debug. Per tutta la serie STM32F1 la stringa è: *DARMSTM.DLL*
- Parameter: in questo campo inseriamo il nome microcontrollore che deve simulare il debugger ovvero -pSTM32F103C8.

Si compila il programma con *Rebuild Target*, si avvia la simulazione e si apre la finestra della seriale USART1 (selezionando la modalità HEX): in basso a destra è visibile la sequenza inviata.

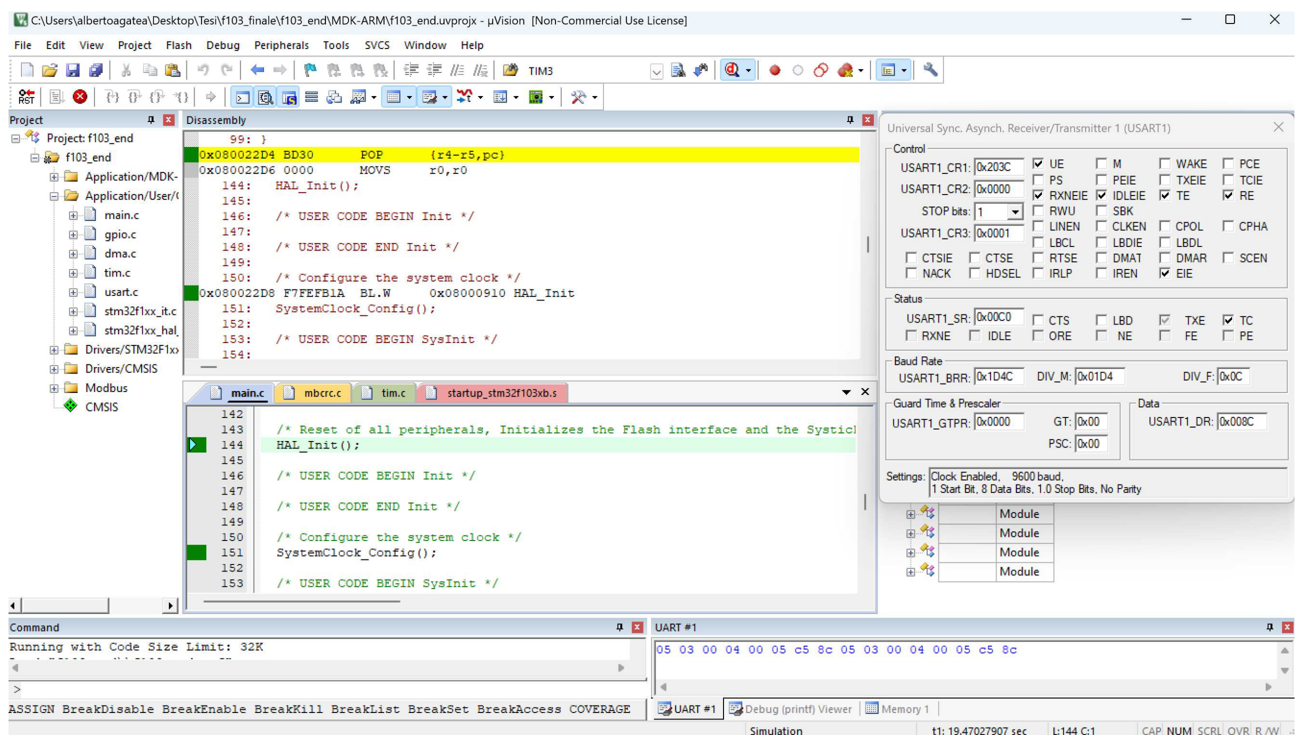

Fig. 3.18: Simulazione Richiesta Master

Nel momento in cui viene avviata la simulazione uVision non crea una COM virtuale alla quale poter inviare i dati trasmessi sulla USART1 simulata.

Per questo motivo, per verificare che la sequenza inviata sia sensata, e mostrare cosa accadrebbe se ci fosse la possibilità di collegare uno slave simulato, si crea una comunicazione virtuale Modbus all'interno del PC.

Per ottenere quanto suddetto è necessario creare un ponte tra 2 COM del computer.

Si installa un apposito software per creare delle COM virtuali e poterle mettere in collegamento fra loro, dove una COM sarà dedicata ad un ipotetico Master ed una ad un ipotetico Slave.

Installando Virtual Serial Port Tools si creano le COM necessarie, in questo caso COM7 per il Master e COM8 per lo Slave.
Una volta messe in ponte si procede all'installazione di Modbus Poll e Modbus Slave i quali appunto simuleranno rispettivamente Master e Slave.

Si configura il Master andando ad inserire i parametri analoghi a quelli implementati precedentemente nel progetto.
Dopo aver configurato lo slave, ed aver inserito dei valori all'interno dei registri, si ottiene il risultato seguente:

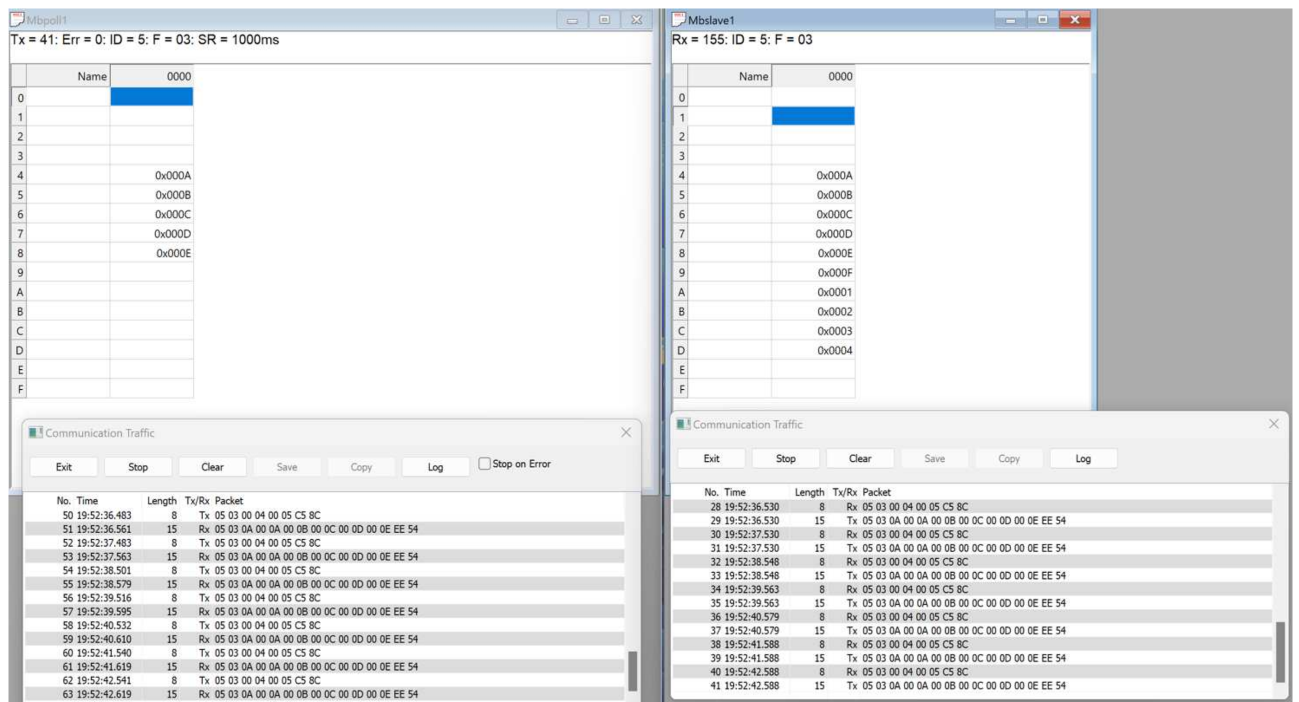

Fig. 3.19: a sx il Master, a dx lo slave

Come visibile nella Fig. 3.19 il pacchetto inviato dal master è lo stesso presente sulla seriale USART1. E' possibile vedere il pacchetto inviato in risposta dallo slave costituito da 15 byte e contenente le informazioni richieste dal Master.
In questo caso il Master esegue un polling sullo slave 5 con intervalli di 1 secondo fra una richiesta e l'altra.

A questo punto è possibile implementare le altre funzioni analizzate nel precedente capitolo e vedere i messaggi che vengono comunicati sulla seriale USART1.
Per una questione di sintesi, tali funzioni non sono qui riportate ma sono implementate nel progetto.

Alla fine di questo documento è presente un *appendice* nel quale viene riportata l'implementazione delle altre funzioni esposte nel capitolo 2, ovvero le funzioni:
01,02,04,05,06.

3.4 Configurazione STM32F334

Questo modello presenta delle caratteristiche più robuste ed è progettato per applicazioni di controllo digitale con elevata precisione temporale, come il controllo di motori o applicazioni di conversioni di potenza.
Per soddisfare queste esigenze implementa diverse periferiche avanzate, fra cui il timer ad alta risoluzione HRTIM che consente di generare impulsi con risoluzione fino a pochi nanosecondi.
Se si volessero inviare messaggi alla velocità massima supportata dal protocollo Modbus, bisognerebbe impostare il Baud Rate a 115200 bit/s e ciò implicherebbe un T_{char} = 95.49 microsecondi. Pertanto anche se non venisse configurato il timer ad alta risoluzione, il risultato sarebbe ugualmente ottenibile da uno degli altri timer "standard" di tipo General Purpose, tranne per la precisione con cui possono essere rilevati i tempi di inattività e quindi per gestire con maggior esattezza eventuali temporizzazioni.

Si configura il microprocessore in maniera analoga a quanto fatto con il modello precedente tranne per il timer ad alta risoluzione del quale riportiamo di seguito la parametrizzazione.

Questa volta si sceglie di inviare i messaggi in Modbus alla massima velocità sostenuta dal protocollo. Dato dunque T_{char} = 95.49 microsecondi, il tempo per fra un messaggio è l'altro deve essere di almeno $T_{3.5}$= 334.25 microsecondi.

HRTIM1 di questo microcontrollore può lavorare fino a 144 MHz. Lavorando a questa frequenza il contatore del timer viene incrementato ogni 6.94 nanosecondi.

Per cui per ottenere un periodo pari a $T_{3.5}$ sono necessari all'incirca 48 156 "tick".
Andando ad impostare questo numero nel campo *Period* si ottiene il periodo desiderato.
Il *Prescaler Ratio* viene impostato di modo che il timer lavori a 144 MHz e la modalità operativa è *Free-running*.
Così facendo il HRTIM1 genera un interrupt o raggiunge l'overflow dopo 334.25 microsecondi, rispettando il requisito di $T_{3.5}$ per il Modbus RTU.

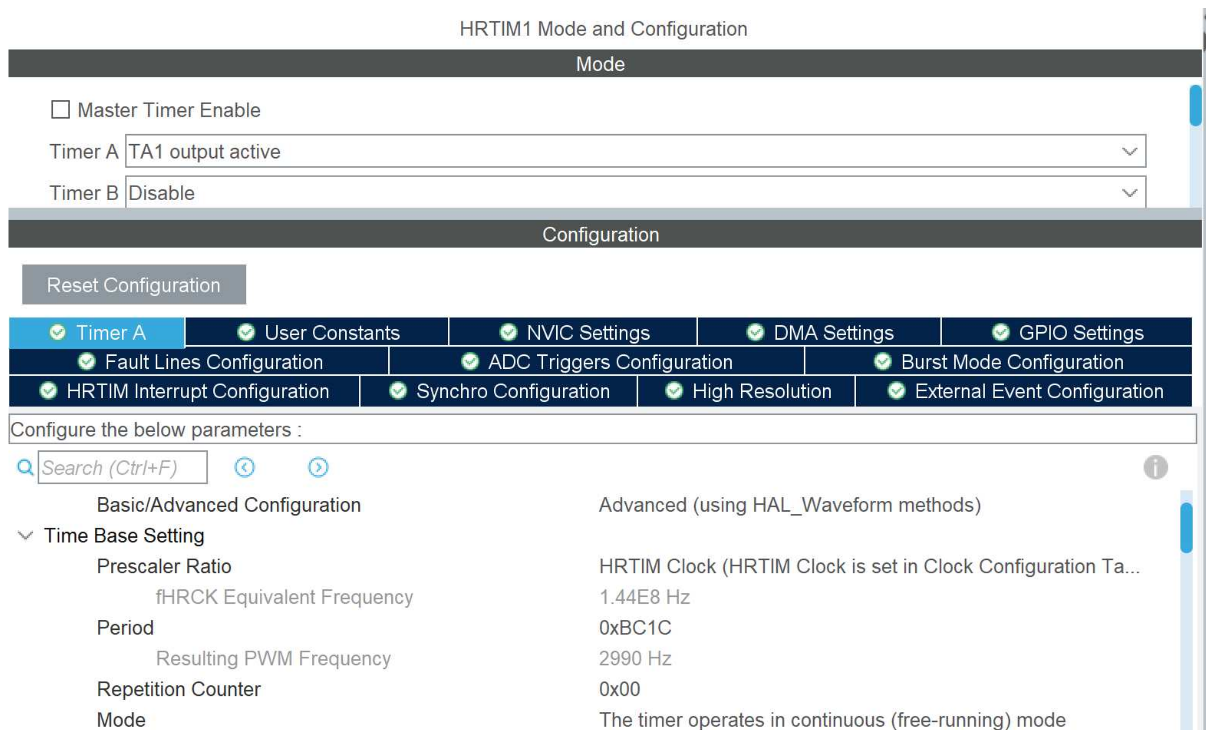

Fig. 3.20: Configurazione timerA

Si aggiunge poi nel DMA HRTIM1, con direzione verso la periferica (USART) per la trasmissione in Modbus e con priorità alta per evitare ritardi di timeout $T_{3.5}$.

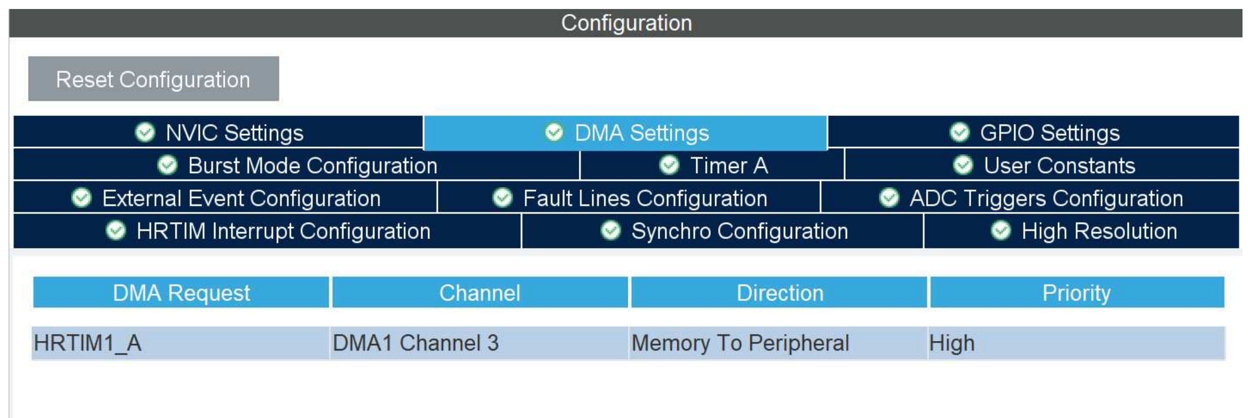

Fig. 3.21

Per l’applicazione in questione, il resto delle specifiche possono essere lasciate come sono di default.

Capitolo 4

Conclusioni

In questo documento, oltre ad aver visionato le caratteristiche principali del protocollo di comunicazione seriale Modbus, si è analizzata la simulazione su uVision 5 di un messaggio inviato sulla USART1 del microcontrollore STM32F103. Inoltre, si è vista la configurazione del STM32F334 per un'ulteriore possibile implementazione come Master all'interno di una comunicazione Modbus RTU tramite RS485.

Quindi entrambi i microcontrollori vanno bene per sostenere una comunicazione di questo tipo nonostante l'F334, implementando il chip Cortex M4, offra caratteristiche superiori sia in termini di velocità che di precisioni rispetto all'F103 che è comunque in grado garantire una buona comunicazione grazie a timer standard e al supporto USART.
Grazie all'uso di software come STM32CubeMX la configurazione dei dispositivi, generalmente complessi come quelli presentati, è semplificata nonostante richieda una conoscenza hardware di base.

Appendice

Funzione 01:

Lettura uscita digitale.
Si modifica il contenuto del messaggio nella maniera seguente:

```
// frame message in modbus RTU è costituito da:
// indirizzo: 1 byte + funzione: 1 byte + dati: fino a 252 bytes + crc: 2 byte

  TxData[0] = 0x05;  // indirizzo dello slave
  TxData[1] = 0x01;  // codice funzione: read coils

  TxData[2] = 0x00;
  TxData[3] = 0x01;
//l'indirizzo del registro è: 00000000 00000001 = 1 + 1 = 2

  TxData[4] = 0;
  TxData[5] = 10;
//il numero dei registri da leggere è: 10 coils, 2 bytes

  uint16_t crc = crc16(TxData, 6);
  TxData[6] = crc&0xFF;   // CRC LOW
  TxData[7] = (crc>>8)&0xFF;  // CRC HIGH
```

Fig.1*

Nella funzione di CallBack:

```
// una volta che lo slave riceve il dato, invia una conferma di avvenuta ricezione al master
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size){

/* andiamo a dividere i byte ricevuti in bit*/
  for (int j=0; j<RxData[2]; j++)
  {
      for (int i=0; i<8; i++)
      {
        Data[indx++] = ((RxData[3+j]>>i)&1);
      }
    }
  }
```

Fig. 2*

Ed infine si cambia in integer l'array dei dati, visto che si tratta di *coils* e cioè di 0 ed 1 e si definisce la variabile *indx* che viene usata come indicato nella figura soprastante.
Ogni volta che si tratta di coils, la funzione di CallBackEvent è bene scriverla come quella nella indicata in fig. 2*. Negli altri casi la funzione di CallBackEvent è analoga a quella indicata per la funzione 03 nel capitolo 3.
Il messaggio sulla USART1:

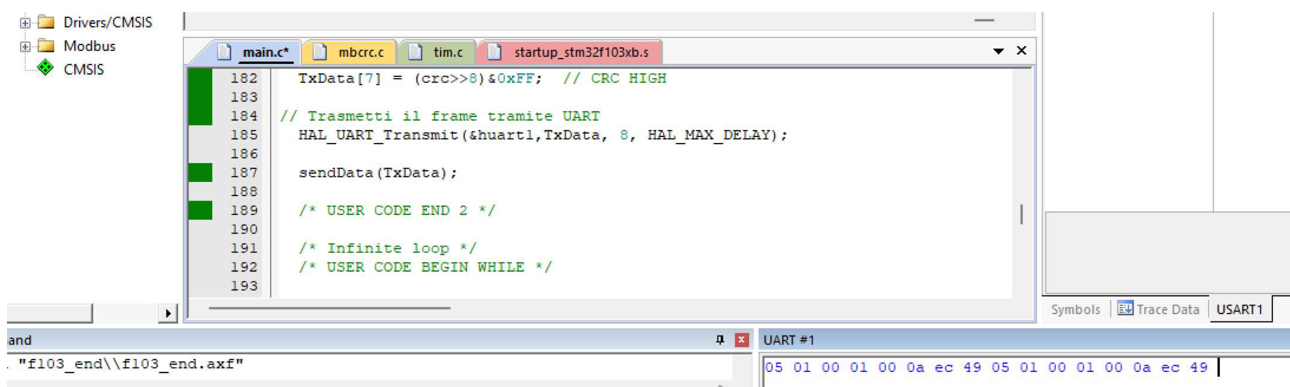

Fig. 3*

A conferma del messaggio mandato ecco lo screen della comunicazione simulata tramite Modbus Poll e Modbus Slave:

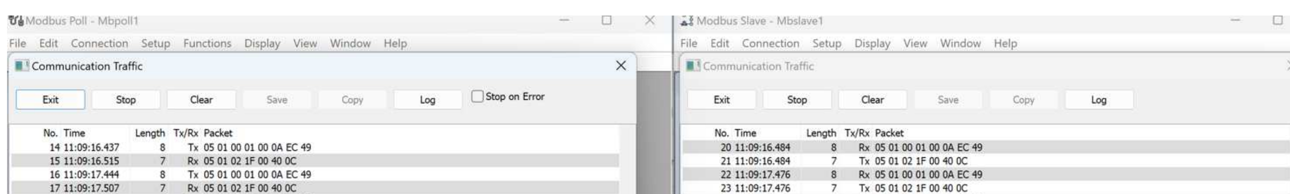

Fig. 4*

Funzione 02:

Per la lettura di ingressi digitali, l'unica cosa da cambiare dalla precedente funzione è il codice che diventa 02.
Il messaggio sulla USART1 è visibile in basso a destra:

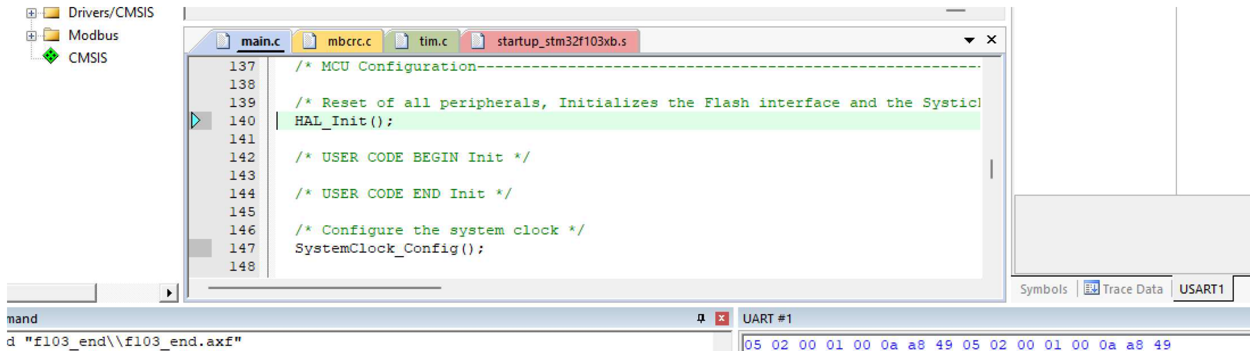

Fig. 5*

Tramite Modbus Poll e Modbus Slave:

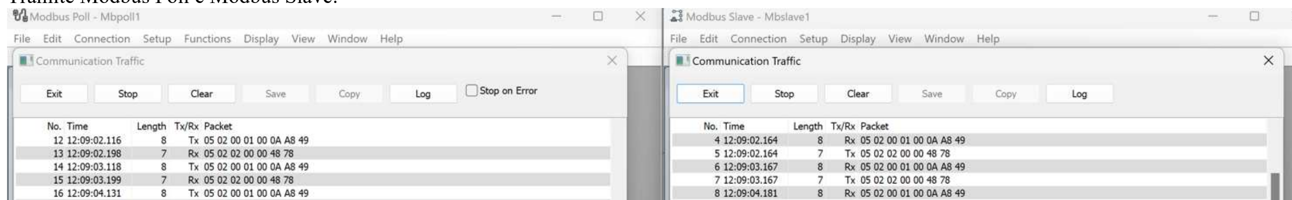

Fig. 6*

Funzione 03:

Lettura uscita analogica.
Si modifica il codice precedente andando ad inserire il codice opportuno della funzione. Il messaggio inviato sulla USART1 e la simulazione con gli appositi software è riportata di seguito.

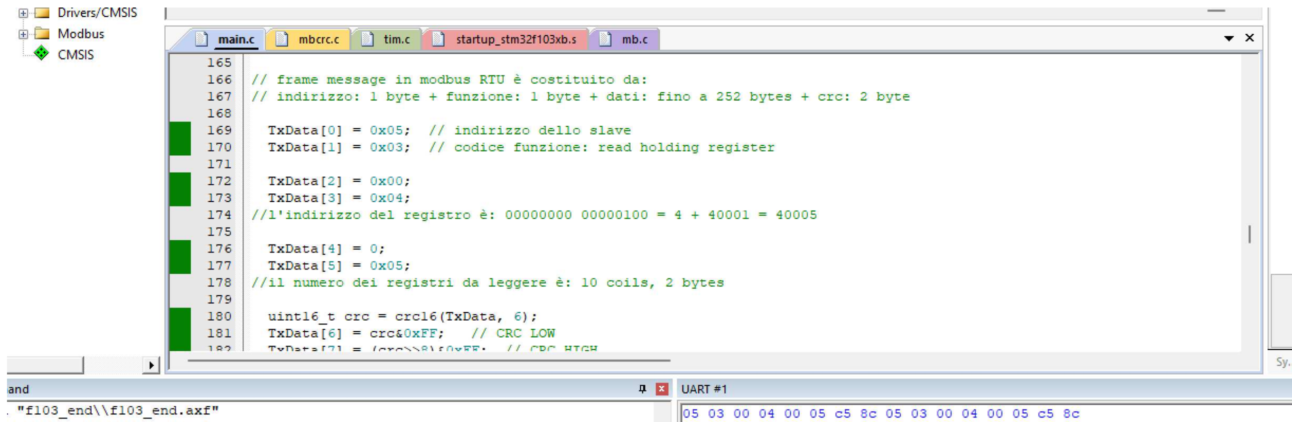

Fig. 7*

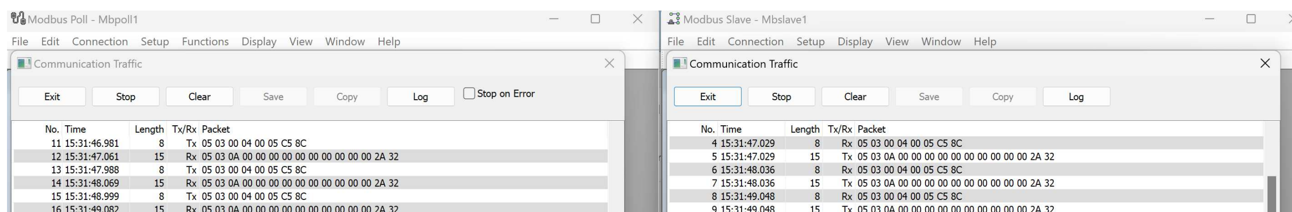

Fig. 8*

Funzione 04:

Lettura ingresso analogico.
Si modifica il codice precedente andando ad inserire il codice opportuno della funzione. Il messaggio inviato sulla USART1 e la simulazione con gli appositi software è riportata di seguito.

```
135   /* USER CODE END 1 */
136
137   /* MCU Configuration--------------------------------------------------------*/
138
139   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
140   HAL_Init();
141
142   /* USER CODE BEGIN Init */
143
144   /* USER CODE END Init */
145
146   /* Configure the system clock */
147   SystemClock_Config();
148
149   /* USER CODE BEGIN SysInit */
150
151   /* USER CODE END SysInit */
```

UART #1

```
05 04 00 01 00 05 60 4d 05 04 00 01 00 05 60 4d
```

Fig. 9*

Modbus Poll - Mbpoll1 — Communication Traffic

No.	Time	Length	Tx/Rx	Packet
30	15:57:59.755	8	Tx	05 04 00 01 00 05 60 4D
31	15:57:59.835	15	Rx	05 04 0A 00 00 00 00 00 00 00 00 00 00 DF F9
32	15:58:00.769	8	Tx	05 04 00 01 00 05 60 4D
33	15:58:00.852	15	Rx	05 04 0A 00 00 00 00 00 00 00 00 00 00 DF F9
34	15:58:01.772	8	Tx	05 04 00 01 00 05 60 4D

Modbus Slave - Mbslave1 — Communication Traffic

No.	Time	Length	Tx/Rx	Packet
6	15:57:59.803	8	Rx	05 04 00 01 00 05 60 4D
7	15:57:59.803	15	Tx	05 04 0A 00 00 00 00 00 00 00 00 00 00 DF F9
8	15:58:00.819	8	Rx	05 04 00 01 00 05 60 4D
9	15:58:00.819	15	Tx	05 04 0A 00 00 00 00 00 00 00 00 00 00 DF F9
10	15:58:01.821	8	Rx	05 04 00 01 00 05 60 4D

Fig. 10*

Funzione 05:

Scrittura uscita digitale.
Si modifica il codice precedente andando ad inserire il codice opportuno della funzione. Il messaggio inviato sulla USART1:

```
135   /* USER CODE END 1 */
136
137   /* MCU Configuration--------------------------------------------------------*/
138
139   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
140   HAL_Init();
141
142   /* USER CODE BEGIN Init */
143
144   /* USER CODE END Init */
145
146   /* Configure the system clock */
147   SystemClock_Config();
148
149   /* USER CODE BEGIN SysInit */
150
151   /* USER CODE END SysInit */
```

UART #1

```
05 05 00 00 ff 00 8d be 05 05 00 00 ff 00 8d be
```

Fig. 11*

I software Modbus Poll e Modbus Slave, nella comunicazione simulata, non danno la possibilità di eseguire funzioni di scrittura sui registri dello Slave.

Per questo motivo, delle funzioni 05 e 06 si riportano solamente i codice inviati sulla USART1

Funzione 06:

Scrittura uscita analogica.
Si modifica il codice precedente andando ad inserire il codice opportuno della funzione. Il messaggio inviato sulla USART1:

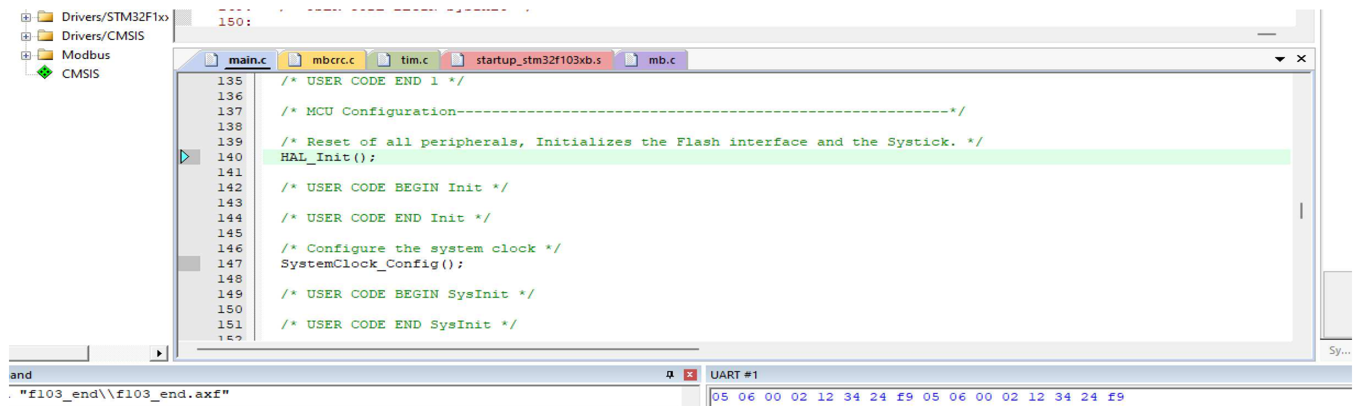

Fig. 12*

Bibliografia e sitografia

[1] Modbus.org

[2] The Everyman's Guide to Modbus ISBN 9781517764685

[3] Arm community

[4] Keil uVision community

[5] uVision user's guide: https://developer.arm.com/documentation/101407/latest/

[6] ControllersTech.com

[7] Modbus Poll User manual