UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'informazione

Corso di laurea in Ingeneria Informatica

Tesi di laurea

LIBRERIA SOFTWARE PER IL CONTROLLO DI UN DISPOSITIVO LASER INDUSTRIALE: PROGETTAZIONE, SVILUPPO, INTEGRAZIONE E TEST

Relatore
Prof. Mirco Rampazzo

Laureando
Andrea Scanu

ANNO ACCADEMICO 2023-2024

DATA DI LAUREA 17/07/2024

INDICE

SOMMARIO

Il presente elaborato illustra una paronamica delle attività e dei risultati ottenuti durante un'esperienza di tirocinio curriculare svolto presso l'azienda DV srl (https://www.dvoptic.com/) terminato con la stesura della tesi di laurea.

In particolare delinea il processo di sviluppo di una libreria di controllo per un dispositivo laser da integrare all'interno di un applicativo già sviluppato dall'azienda richiedente e implementata tramite il linguaggio di programmazione Delphi.

La libreria si avvale di diversi oggetti per facilitare lo scambio di messaggi tramite seriale con il laser, consentendo la ricezione di informazioni da esso e la gestione dell'emissione dell'onda laser.

Vengono forniti dettagli sulla struttura statica di ogni elemento della libreria, in particolare le informazioni e le funzionalità di cui ogni elemento deve disporre per il corretto funzionamento, la logica di funzionamento delle sue funzionalità principali e le relazioni intercorrenti tra gli elementi.

In conclusione viene esaminato l'impiego pratico della libreria all'interno dell'applicativo, sottolineando il ruolo svolto nel consentire la comunicazione efficace, in termini di agevolezza per l'utente finale, con il laser.

INTRODUZIONE

DV è un laboratorio tecnologico che progetta e realizza strumentazione spettroscopica e laser per applicazioni scientifiche e industriali. Oltre a produrre i propri strumenti standard, DV fornisce ai clienti specifiche soluzioni software e hardware e assistenza. DV rappresenta, per il mercato italiano, una delle più importanti aziende produttrici di strumenti spettroscopici.
Durante il tirocinio si è intervenuto sulla modifica del software sviluppato dalla DV allo scopo di integrare il supporto ad un modello di laser fornito dall'azienda Wuhan Raycus (https://en.raycuslaser.com), impiegato in un macchinario dedicato alla pulizia dei rulli Anilox.

Nella stampa l' Anilox è un metodo utilizzato per fornire una quantità di inchiostro su una lastra di stampa flessografica.
Un rullo Anilox è un cilindro rigido, solitamente costituito da un'anima di acciaio o alluminio rivestita da una ceramica industriale, la cui superficie è incisa con milioni di fossette molto sottili, note come celle Anilox.
Nel processo di stampa, il rullo Anilox viene rivestito con uno strato di inchiostro che viene poi trasferito sulle porzioni in rilievo della lastra di stampa.
Il numero, la dimensione e la geometria delle celle Anilox variano e determinano la quantità di inchiostro che il rullo anilox invia alla lastra. Dimensioni più ridotte consentono la creazione di pattern più precisi e dettagliati nel disegno desiderato.

Parametri caratteristici:

	U.D.M.	**Descrizione**
Lineatura	c/cm cpi	Indica il numero di celle presenti per unità di area
Volume Nominale	cm^3/m^2 BCM/in^2	Indica il volume di inchiostro impiegato per unità di area

Il macchinario attraverso l'utilizzo di un laser automatizza il processo di pulizia dei rulli Anilox eliminandone l'inchiostro in eccesso grazie ad un insieme di periferiche che muovono in autonomia il rullo e configurando i parametri relativi alle dimensioni del rullo, la configurazione del laser e le varie velocità con cui il laser passerà tra le varie celle.

APPLICATIVO EVOLUX

L’applicativo Evolux viene eseguito in ambiente Windows 10/11 64bit ed è sviluppato tramite il linguaggio di programmazione Delphi.

Delphi, detto anche ‘Pascal a oggetti’, è un derivato del linguaggio di programmazione Pascal ed è principalmente utilizzato per lo sviluppo veloce di interfaccie utente, interfacce a database, dispositivi hardware ecc...
E’ dotato di un proprio ambiente di sviluppo con compilatore annesso, un editor di codice con funzionalità integrate di refactoring e controllo sintattico evoluto, uno strumento per la realizzazione visuale delle interfacce grafiche e il supporto per il plugin di terze parti.

I file sorgente delle applicazioni vengono salvati in dei file aventi estensione .pas, tali file vengono suddivisi nelle sezioni **interface**, riportante l'interfaccia del programma, e **implementation**, riportante la logica implementata in esse.
Se e presente un'interfaccia grafica essa viene salvata in un file separato riportante lo stesso identificativo del .pas e avente estensione .dfm.

L’applicativo viene eseguito suddividendo le sue attività in più thread, il thread principale è adibito alla gestione dell’interfaccia utente mentre i rimanenti sono adibiti al monitoraggio periodico del corretto funzionamento delle varie periferiche (laser, sensori, microscopio ecc…) che deve essere garantito per un funzionamento corretto del macchinario.

La UI dell’ applicativo è suddivisa nelle seguenti sezioni:

- **HOME**: Sezione principale dell’ applicativo, da questo form è possibile monitorare lo svolgimento della pulizia automatica del rullo
- **PROGRAMMI**: Sezione di gestione e selezione delle ricette di parametrizzazione delle pulizie
- **NUOVO PROGRAMMA**: Sezione di creazione delle ricette di parametrizzazione delle pulizie
- **MANUALE**: Sezione di diagnostica designata all’ installatore della macchina per verificarne il corretto funzionamento, permette il controllo manuale delle periferiche installate nel macchinario
- **DIAGNOSTICA**: Sezione di controllo dei parametri delle periferiche
- **DATABASE**: Sezione di controllo delle precedenti pulizie e dei rulli anilox salvati nel database dell’applicativo
- **UTENTI**: Sezione di log-in e gestione utenti
- **IMPOSTAZIONI**: Sezione di impostazione delle variabili di sistema relative alla UI, alle periferiche ecc…
- **SUPPORTO**: Sezione riportante i contatti dell’azienda

Molteplici componenti grafiche dell'applicativo segue il design pattern Model-View-Controller, in cui il modello è rappresentato dalle variabili fornite alla componente e aggiorna la view ogni volta che viene modificato, la View è formata dalla componente grafica dei form e varia in base allo stato del Model e il Controller viene chiamato in base agli eventi scaturiti dall'interazione dell'utente con l'UI o da altri thread e si occupa di modificare il modello come gli viene richiesto.

La comunicazione tra i vari thread dell'applicativo è gestita sulla base del design pattern publish – subscribe, che raccomanda la divisione dei messaggi che possono essere pubblicati dai vari thread in classi distinte, e ogni componente dell'applicativo che ha bisogno di ricevere messaggi relativi ad una classe specifica ha la possibilità di iscriversi alla ricezione del tipo specifico di messaggio.

LASER RAYCUS

Modello di laser Raycus serie MX, https://en.raycuslaser.com

Il laser Raycus Serie-MX a fibra pulsata utilizza la struttura main oscillator power amplifier (MOPA) in cui il laser emesso viene amplificato in potenza da un amplificatore in fibra ottica.

Per ogni modello il dispositivo viene fornito di un insieme di configurazioni associate alla durata di un singolo impulso (compreso l'impulso continuo) da cui viene poi fissato un intervallo di frequenze utilizzabili.

L'energia di un singolo impulso del laser dipende dal rapporto tra la sua potenza e la frequenza impiegata, fino al massimo di energia erogabile dipendente fisicamente dalla durata di singolo impulso scelta e oltre al quale diventa costante al diminuire della frequenza.

100 W					
	τ [ns]	PRR_{min} [kHz]	PRR_0 [kHz]	PRR_{max} [kHz]	$E_{\text{Puls, max}}$ [mJ]
1	10	1	1000	2000	0.1
2	20	1	500	2000	0.2
3	30	1	330	1000	0.3
4	60	1	200	1000	0.5
5	100	1	125	1000	0.8
6	200	1	80	500	1.25
7	250	1	72	500	1.4
8	350	1	66	400	1.5
9	600	–	–	–	CW

Esempio di valori disponibili con modello da 100W

Il computer si interfaccia con il dispositivo tramite una comunicazione con seriale, in particolare ogni comando viene codificato tramite una sequenza di byte e se viene eseguito correttamente dal dispositivo esso ritorna al computer una sequenza di risposta, sia le sequenze mandate che quelle ricevute presentano questa interfaccia generale:

Parametri	**Numero di Byte**	**Valori**	**Descrizione**
Data header	4	FE FE FE 68	Caratteri di inizio sequenza
Address	2	FF FF	Valore fisso
Command word	1		Byte di selezione del comando
Alternate parameter	1		Dipendente dal comando richiesto
Data length	2		N byte inseriti come dati del comando
Data	Valore di data length		Dipendente dal comando richiesto
Check code	2		CRC16 di verifica ricezione corretta
Data tail	1	55	Carattere di fine sequenza

Il dispositivo inizia a leggere i byte che riceve da quando identifica la stringa di Data Header e terminata la ricezione del comando verifica che la codifica CRC16 dei byte a partire dagli associati all'indirizzo a quelli relativi ai dati siano conrrispondenti ai due byte di Check Code ricevuti dalla seriale, in caso affermativo esegue il comando ricevuto se è valido e manda alla seriale la stringa di risposta.

Il laser supporta i seguenti comandi:

- **LETTURA DEI PARAMETRI:** permette all'utente di leggere vari parametri del dispositivo
- **ACCENSIONE/SPEGNIMENTO E SELEZIONE DEI PARAMETRI:** permette all'utente di selezionare pulse width, frequenza e potenza dell'onda all'inizio dell'emissione del laser o di terminarne l'emissione
- **EMISSIONE LUCE ROSSA:** il dispositivo è dotato anche di un led rosso utile per poterlo posizionare correttamente prima dell'utilizzo
- **SERIAL NUMBER/SECRET KEY:** permette di ottenere via software il numero di seriale e la secret key del dispositivo
- **FIRMWARE:** permette di ottenere via software il firmware utilizzato dal dispositivo

LIBRERIA RAYCUS

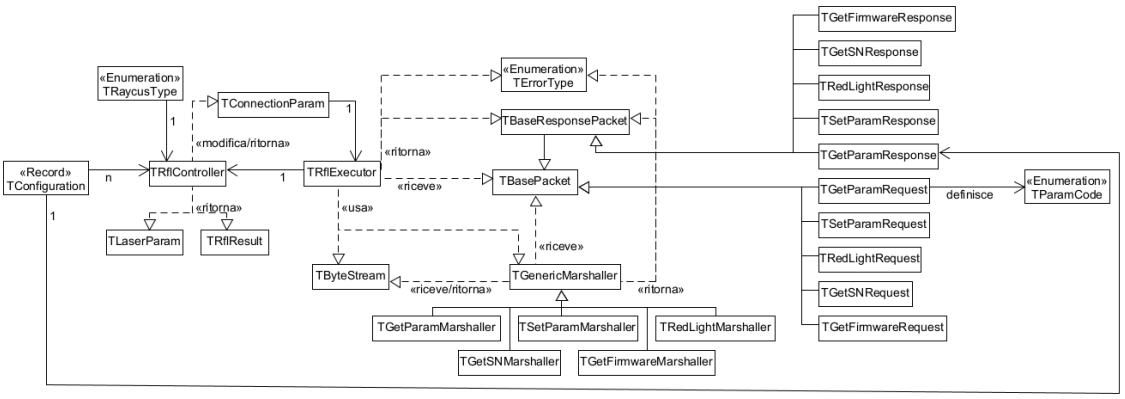
Per implementare la comunicazione tra il computer e il dispositivo sono stati implementati diversi elementi che sono state suddivise nei seguenti file:

Diagramma UML dei package

La libreria è stata sviluppata secondo il principio della progettazione modulare, secondo cui ogni componente collabora con gli altri ed è incaricato di svolgere un compito preciso nell'elaborazione delle richieste da comunicare al dispositivo.

In particolare il package *types* contiene diversi contenitori di informazioni necessarie per il collegamento col dispositivo e l'elaborazione delle richieste,
il package *packet* contiene diversi contenitori di informazioni relative alle singole richieste disponibili,
il package *marshaller* contiene diverse classi adibite alla conversione tra le informazioni richieste e ritornate dal dispositivo in base alla richiesta che si intende eseguire e il relativo stream di byte,
il package *executor* contiene la classe adibita allo scambio di informazioni tra il computer e il dispositivo,
il package *controller* contiene la classe utilizzata come intermediario tra l'utente e il dispositivo.

L'utilità di ogni elemento della libreria verrà approfondita nelle pagine successive del documento.

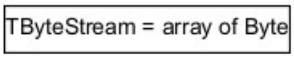


Schema UML di classe della libreria

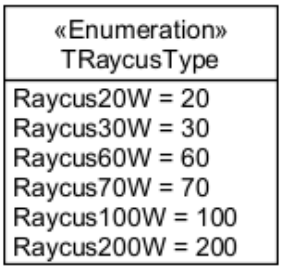
INCAPSULAMENTO DEI DATI

Le informazioni necessarie alla comunicazione tra il dispositivo e il computer sono incapsulate in diversi elementi definiti all'interno dei package *types* e *packet.*

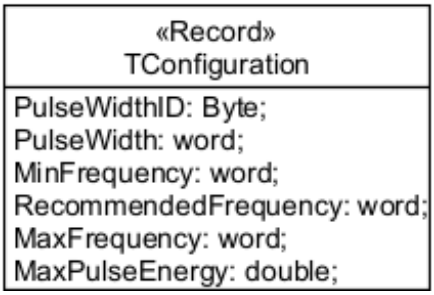
All' interno del package *types* sono definiti i seguenti elementi utilizzati all'interno delle altre classi (mostrate con i *relativi diagrammi UML di classe*):



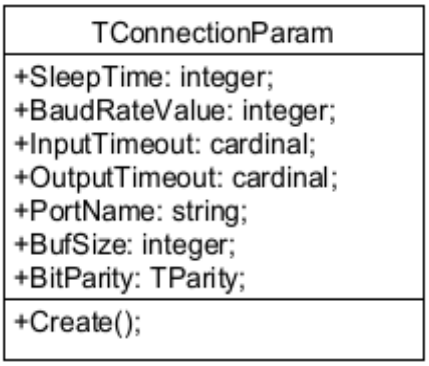
TByteStream viene utilizzata come ridenominazione di un array di byte per rappresentare le sequenze di byte utilizzate per interfacciarsi col dispositivo



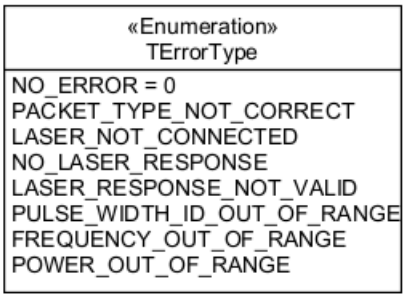
TRaycusType è un'enumerazione utilizzata per distinguere il modello di laser interfacciato in base alla sua potenza espressa in Watt, distinzione necessaria per applicare i giusti vincoli di frequenza e pulse width relativi al singolo dispositivo



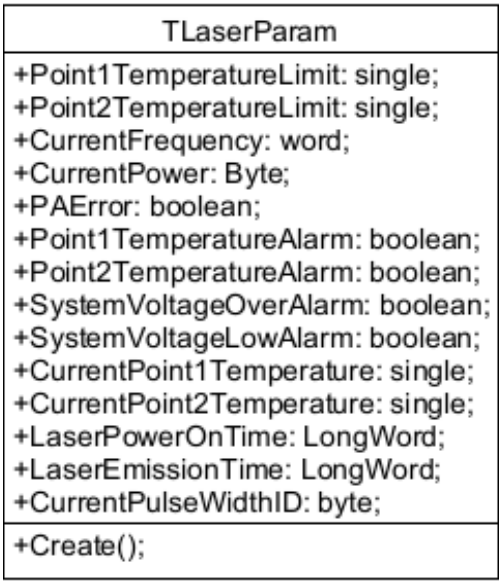
TConfiguration è un record utilizzato per creare gli array con i parametri di configurazione relativi ai modelli di laser



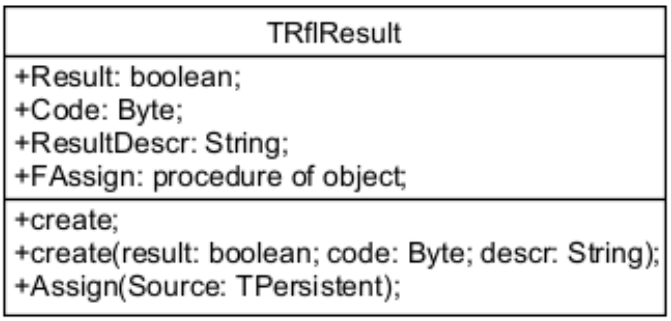
TConnectionParam è una classe utilizzata come contenitore dei parametri definiti per impostare la connessione con il laser



TErrorType è un'enumerazione rappresentante l'insieme dei possibili errori che possono presentarsi alla richiesta di un comando al dispositivo da parte di un oggetto TRflController

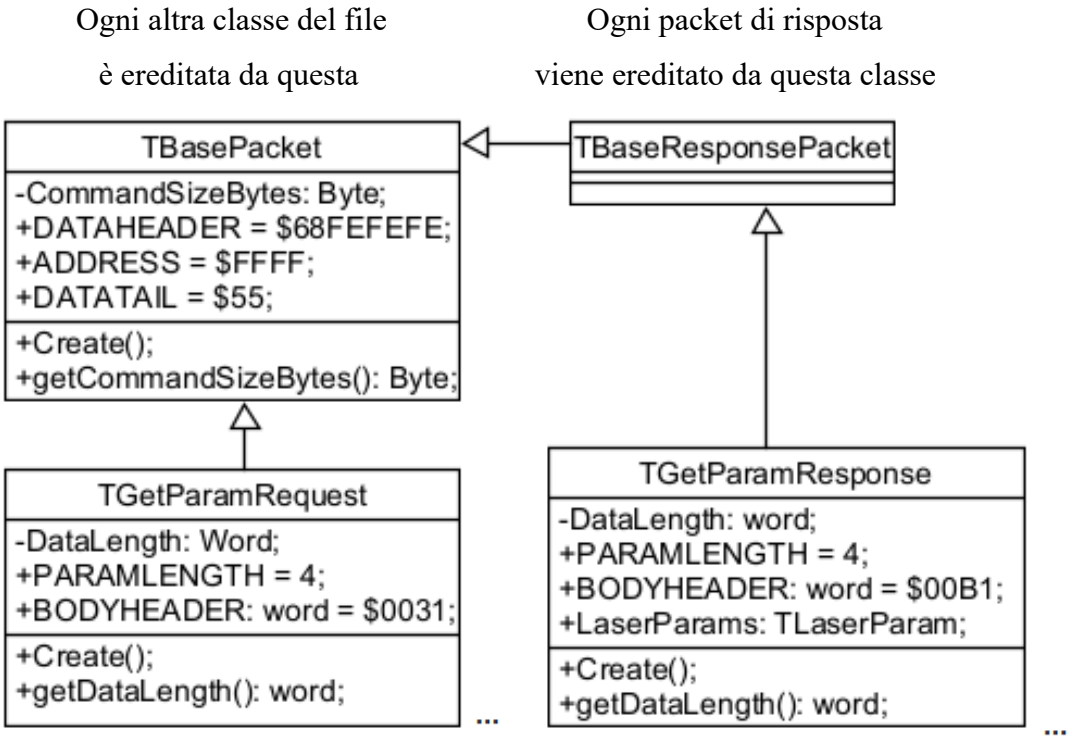


TLaserParam è una classe utilizzata come contenitore dei parametri ritornati dal dispositivo alla chiamata del relativo comando



TRflResult è una classe rappresentante l'esito di un metodo eseguito da un oggetto TRflController per dialogare con il dispositivo, contenente l'esito positivo o meno della richiesta, il relativo codice di errore e una sua descrizione come stringa.

Nel package *packet* vengono implementate un insieme di classi utilizzate per rappresentare i valori di input necessari al marshaller per codificare l'array di byte corrispondente o per verificare la coerenza dell'array ricevuto in risposta e salvare i valori della risposta nel caso in cui sia coerente.



Diagrammi UML di classe di alcune classi del package

CONVERSIONE DATI/BYTESTREAM

La conversione tra i dati salvati all'interno degli oggetti definiti nella classe *packet* e i relativi byteStream viene definita all'interno del packet *marshaller*.

Nel packet *marshaller* vengono definiti un'insieme di classi adibite a convertire i dati salvati negli oggetti request nei byteStream corrispondenti per essere trasmessi al dispositivo e a convertire i byteStream ricevuti dal dispositivo nei valori di risposta corrispondenti, permettendo così di tenere separati i comandi che l'utente vuole impartire dalla sua corrispondente sequenza di byte che legge il dispositivo.

Tutte le classi marshaller vengono derivate da questa classe:

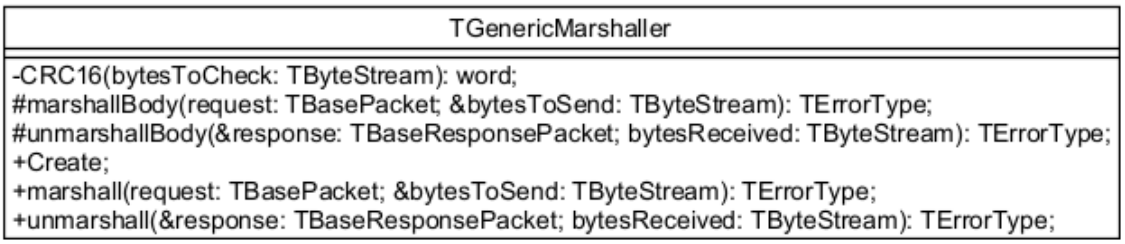


Diagramma UML di classe della classe TGenericMarshaller

Essa implementa l'algoritmo CRC16 di controllo dei byte relativi e i metodi marshall e unmarshall che svolgono le operazioni generali relative a tutti i comandi.

Tutti i marshaller che derivano da essa sovrascrivono i metodi marshalBody e unmarshalBody per essere coerenti con il comando per cui vengono impiegati, di seguito si descrive l'uso generico delle due funzioni del marshaller:

MARSHALL:

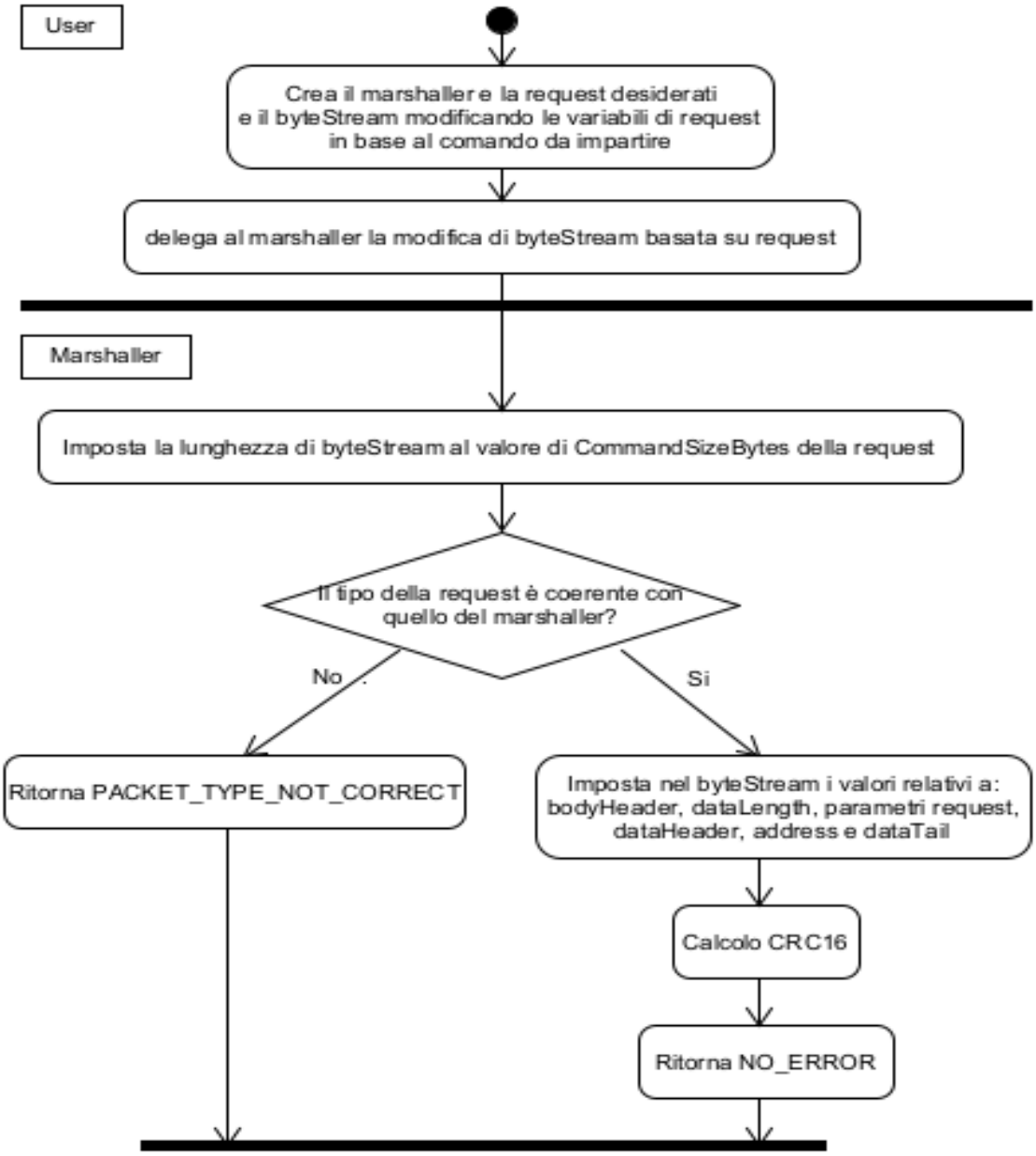


Diagramma UML di attività del metodo Marshall

L'utente chiama la funzione marshall() del marshaller passando in input la request con i parametri relativi al comando che si vuole impartire e ricevendo come output un enum TErrorType che indica l'esito della funzione e se conclusa con successo un TByteStream con la codifica in byte della richiesta pronta per essere mandata al dispositivo.

In particolare viene prima impostata la lunghezza del TByteStream (il numero di byte salvati in esso) col valore della variabile commandSizeBytes della request, poi viene chiamato il metodo marshallBody() specifico del marshaller utilizzato che inizialmente controlla che il tipo della request sia coerente con quello del marshaller e in caso negativo termina la funzione ritornando il TErrorType PACKET_TYPE_NOT_CORRECT, altrimenti vengono caricati i valori relativi a command word, alternate parameter, dataLength e i parametri specifici della richiesta nei byte del TByteStream relativi e la funzione ritorna NO_ERROR.

Il metodo marshall() a sua volta termina se riceve da marshallbody() un valore diverso da NO_ERROR, altrimenti carica nel TByteStream i valori relativi a dataHeader, address e dataTail ed esegue l'algoritmo di codifica CRC16 sui byte necessari per poi caricare anche il risultato dell'algoritmo nel TByteStream e infine termina ritornando NO_ERROR.

```
function TGenericMarshaller.marshall(const request: TBasePacket; var bytesToSend: TByteStream): TErrorType;
var
   pBytesToSend: pLongWord;
begin
   SetLength(bytesToSend, request.CommandSizeBytes);

   Result := self.marshallBody(request, bytesToSend);
   if not(Result = TErrorType.NO_ERROR) then exit;

   pBytesToSend := Addr(bytesToSend[0]);
   pBytesToSend^ := request.DATAHEADER;
   Inc(pBytesToSend);
   pWord(pBytesToSend)^ := request.ADDRESS;
   bytesToSend[request.CommandSizeBytes - 1] := request.DATATAIL;

   pWord(@bytesToSend[request.CommandSizeBytes - 3])^ := CRC16(bytesToSend);

   Result := TErrorType.NO_ERROR;
end;
```

Implementazione in Delphi del metodo marshall

```
function TSetParamMarshaller.marshallBody(const request: TBasePacket; var bytesToSend: TByteStream): TErrorType;
var
   pBytesToSend: pByte;
   requestT: TSetParamRequest;
begin
   if not(request is TSetParamRequest) then
   begin
      Result := TErrorType.PACKET_TYPE_NOT_CORRECT;
      Exit;
   end;
   requestT := request as TSetParamRequest;

   bytesToSend[6] := requestT.getEmissionCommand;
   if requestT.getEmission then
   begin
      bytesToSend[7] := requestT.getPulseWidthID;
      pBytesToSend := Addr(bytesToSend[8]);
      pWord(pBytesToSend)^ := requestT.getDataLength SHL 8;
      pBytesToSend := Addr(bytesToSend[10]);
      pLongWord(pBytesToSend)^ := LongWord(requestT.getFrequency);
      pBytesToSend := Addr(bytesToSend[14]);
      pLongWord(pBytesToSend)^ := LongWord(requestT.getPower);
   end;

   Result := NO_ERROR;
end;
```

Implementazione di esempio di un metodo marshallBody

UNMARSHALL:

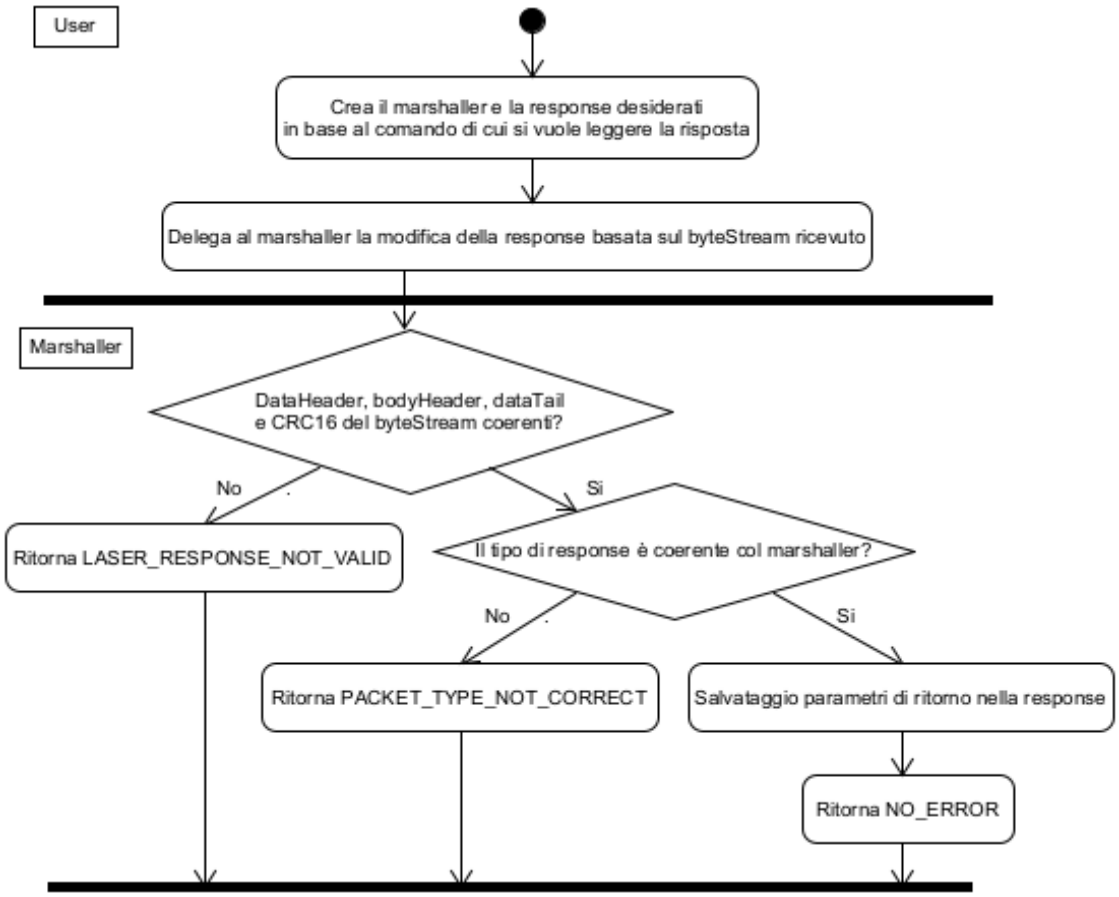


Diagramma UML di attività del metodo Unmarshall

L'utente chiama la funzione unmarshall() del marshaller passando in input il TByteStream ricevuto dal dispositivo in risposta ad un comando impartito e ricevendo come output un enum TErrorType che indica l'esito della funzione e se conclusa con successo una response con i valori di ritorno del comando.

In particolare viene prima controllato che i primi byte del TByteStream ricevuto corrispondano al dataHeader e gli ultimi corrispondano al dataTail adeguati e che la codifica CRC16 dei parametri relativi eseguita via software sia uguale a quella presente all'interno del TByteStream, in caso negativo la funzione termina ritornando il TErrorType LASER_RESPONSE_NOT_VALID, altrimenti ritorna l'output dell'unmarshallBody relativo al comando specifico.

L'unmarshallBody ritorna il TErrorType PACKET_TYPE_NOT_CORRECT se il tipo della response è incoerente con quello del marshaller, altrimenti se il command word presente nel TByteStream ricevuto è incoerente con il comando richiesto viene ritornato

il TErrorType LASER_RESPONSE_NOT_VALID, altrimenti si considera valida la risposta del laser, quindi vengono salvati nella response gli eventuali valori di ritorno del laser e viene dato in output il TErrorType NO_ERROR.

```
function TGenericMarshaller.unmarshall(var response: TBaseResponsePacket; const bytesReceived: TByteStream): TErrorType;
begin
   if (pLongWord(@bytesReceived[0])^ <> response.DATAHEADER) or
      (pWord(@bytesReceived[length(bytesReceived) - 3])^ <> CRC16(bytesReceived)) or
      (bytesReceived[length(bytesReceived) - 1] <> response.DATATAIL) then
   begin
      Result := TErrorType.LASER_RESPONSE_NOT_VALID;
      Exit;
   end;

   Result := unmarshallBody(response, bytesReceived);
end;
```

Implementazione in Delphi del metodo unmarshall

```
function TSetParamMarshaller.unmarshallBody(var response: TBaseResponsePacket; const bytesReceived: TByteStream)
   : TErrorType;
var
   responseT: TSetParamResponse;
begin
   if not(response is TSetParamResponse) then
   begin
      Result := TErrorType.PACKET_TYPE_NOT_CORRECT;
      Exit;
   end;
   responseT := response as TSetParamResponse;

   if bytesReceived[6] = responseT.ONCONFIRMCODE then
      responseT.TurnedOn := true
   else if bytesReceived[6] = responseT.OFFCONFIRMCODE then
      responseT.TurnedOn := false
   else
   begin
      Result := TErrorType.LASER_RESPONSE_NOT_VALID;
      Exit;
   end;
   responseT.PulseWidthID := bytesReceived[7];
   Result := NO_ERROR;
end;
```

Implementazione di esempio del metodo unmarshallBody

COMUNICAZIONE CON IL DISPOSITIVO

L’invio e la ricezione dei byteStream col dispositivo viene gestita all’interno del package *executor*.

All'interno di questo file viene implementata la classe TRflExecutor, incaricata di gestire il collegamento con il laser, di mandare ad esso i byteStream relativi ai comandi desiderati ed elaborare la risposta relativa.

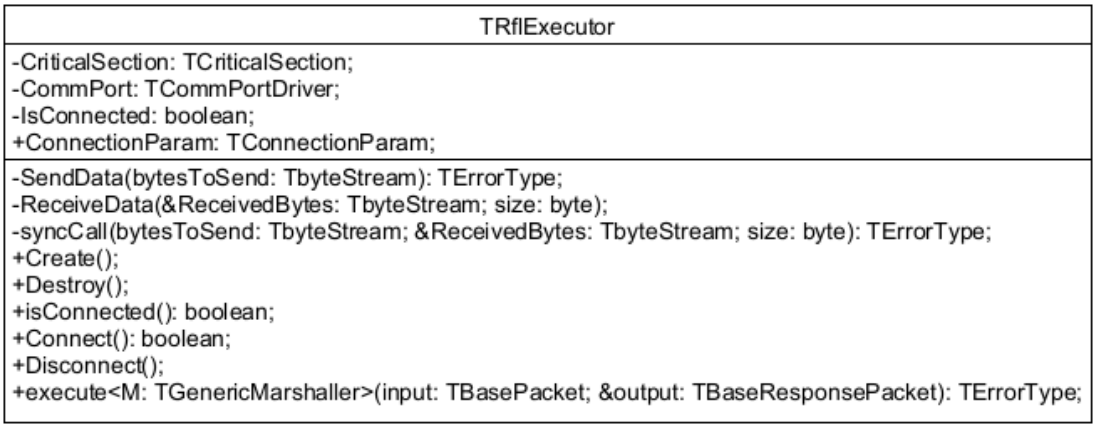


Diagramma UML di classe della classe TRflExecutor

All'interno incapsula le seguenti variabili d'istanza:

- CriticalSection: garantisce che ad ogni richiesta mandata al laser venga associata la risposta corrispondente nel caso in cui più thread utilizzino la stessa istanza dell'executor, dato che la seriale è una risorsa esclusiva
- CommPort: oggetto incaricato di collegarsi alla seriale, inviare i TByteStream desiderati e ricevere i TByteStream mandati dal dispositivo
- IsConnected: indicatore per segnalare se l'executor è connesso o meno al dispositivo
- ConnectionParam: insieme dei parametri usati per impostare la connessione con il dispositivo

All'interno di ConnectionParam vengono incapsulate le seguenti variabili:

Parametro	**Descrizione**
SleepTime	Tempo di attesa tra l'invio e la lettura della risposta
PortName	Riferimento alla porta a cui è collegato il laser
Baud Rate	Velocità di trasmissione
Buffer Size	Dimensione massima buffer
Input Time Out	Tempo di attesa post-invio
Output Time Out	Tempo di attesa ricezione
Bit Parity	Strumento di prevenzione errori

Il metodo più importante della classe TRflExecutor è il metodo Execute:

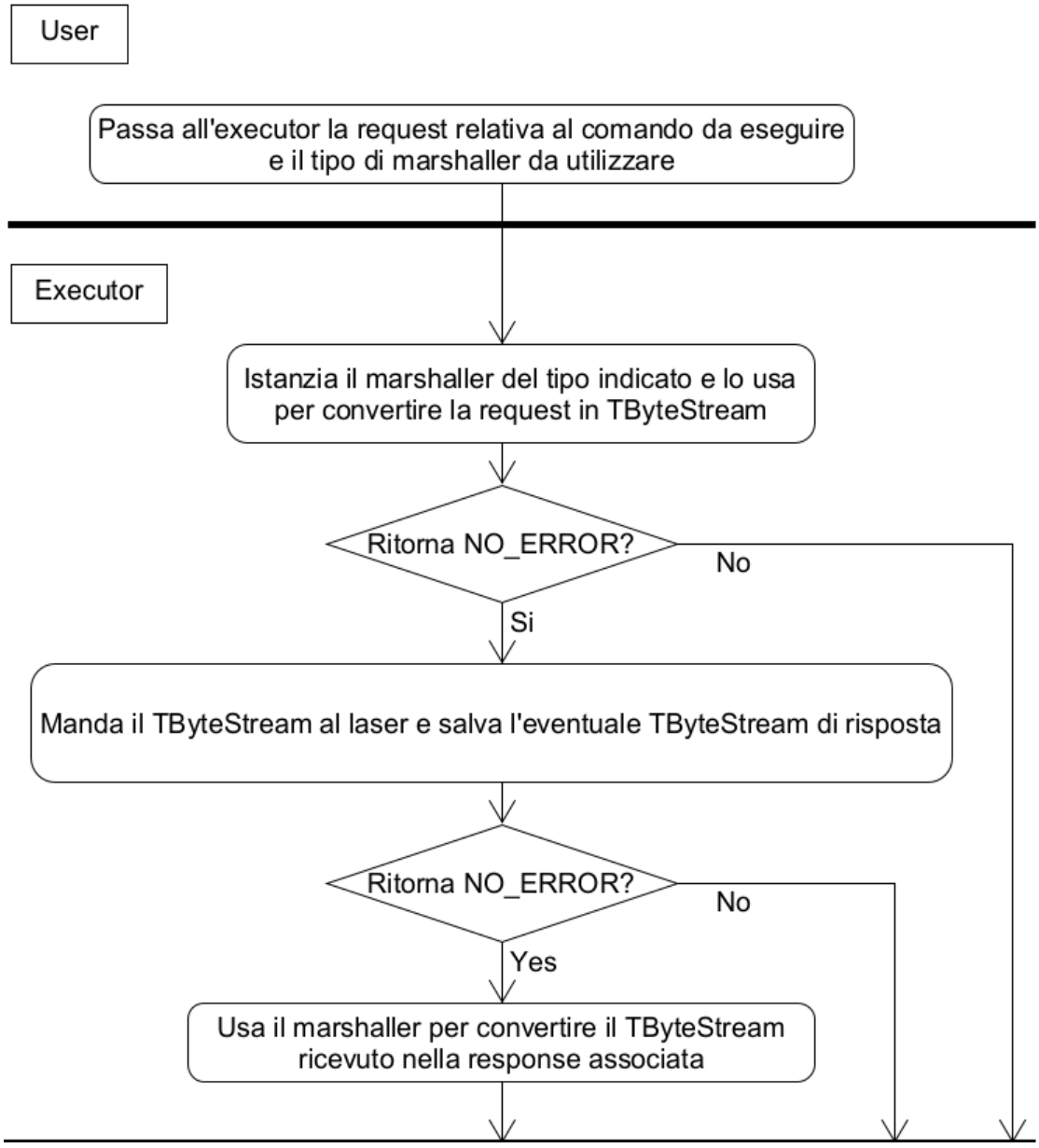


Diagramma UML di attività del metodo Execute

Il metodo riceve come input la request rappresentante il comando da impartire e il tipo di marshaller da utilizzare per convertirla in TByteStream e ritorna come output un TErrorType che indica l'esito dell'operazione e se conclusa con successo la response rappresentante i valori che il laser ritorna dopo l'elaborazione della richiesta.

In particolare all'inizio istanzia il marshaller del tipo desiderato e lo incarica di convertire la request nel TByteStream corrispondente e se il metodo marshall ritorna un TErrorType diverso da NO_ERROR termina la funzione ritornando il relativo errore, altrimenti accede alla CriticalSection per garantire che non vengano effettuate richieste multiple al dispositivo nello stesso momento.

All'interno della CriticalSection viene chiamato il metodo privato SyncCall che prende come input il TByteStream da mandare al laser e il numero di byte che ci si aspetta di ricevere come risposta (non necessario ma utile per non allocare memoria superflua) e ritorna come output il TByteStream ricevuto dal laser e il TErrorType che indica l'esito dell'operazione.

All'interno della SyncCall viene prima chiamato il metodo privato SendData che prende come input il TByteStream da mandare, incaricando l'oggetto CommPort di mandare un byte alla volta, e ritorna LASER_NOT_CONNECTED se il laser non è collegato o NO_ERROR se l'operazione va a buon fine.

Se SendData ritorna LASER_NOT_CONNECTED SyncCall termina ritornando lo stesso errore, altrimenti aspetta per il tempo indicato dalla variabile ConnectionParam.SleepTime e poi chiama il metodo privato ReceiveData che prende come input il numero di byte che ci si aspetta di ricevere e da come output il TByteStream ricevuto dal laser.

ReceiveData prima alloca memoria ad un puntatore temporaneo pari all'input ricevuto e poi incarica la CommPort di salvare nella memoria allocata i byte ricevuti dalla seriale, poi copia i valori salvati in questa memoria all'interno del TByteStream da ritornare in output e infine dealloca la memoria temporanea.

Il metodo SyncCall infine verifica la lunghezza del TByteStream ricevuto e se è pari a 0 ritorna NO_LASER_RESPONSE altrimenti ritorna NO_ERROR.

A questo punto il metodo execute è libero di uscire dalla CriticalSection, poi ritorna direttamente l'errore ritornato da SyncCall se diverso da NO_ERROR, altrimenti incarica il marshaller di convertire il TByteStream ricevuto nella response relativa ritornando l'errore dato dal marshaller.

```
function TRflExecutor.execute<M>(const input: TBasePacket; var output: TBaseResponsePacket): TErrorType;
var
   marshaller: M;
   bytesToSend, ReceivedBytes: TbyteStream;
begin
   try
      marshaller := M.Create();
      bytesToSend := nil;
      ReceivedBytes := nil;

      result := marshaller.marshall(input, bytesToSend);

      if result <> TErrorType.NO_ERROR then exit;

      FCriticalSection.Acquire;
      result := self.syncCall(bytesToSend, ReceivedBytes, output.CommandSizeBytes);
      FCriticalSection.Release;

      if result <> TErrorType.NO_ERROR then exit;

      result := marshaller.unmarshall(output, ReceivedBytes);

   finally
      marshaller.destroy;
   end;

end;
```

Implementazione in Delphi del metodo Execute

```
function TRflExecutor.syncCall(const bytesToSend: TbyteStream; var ReceivedBytes: TbyteStream; const size: byte): TErrorType;
begin
   result := SendData(bytesToSend);
   if result <> TErrorType.NO_ERROR then
   begin
      FCommPort.FlushBuffers(True, True);
      exit;
   end;
   sleep(FConnectionParam.SleepTime);
   ReceiveData(ReceivedBytes, size);
   if length(ReceivedBytes) = 0 then
      result := NO_LASER_RESPONSE;
end;
```

Implementazione in Delphi del metodo SyncCall

```
function TRflExecutor.SendData(const bytesToSend: TbyteStream): TErrorType;
var
   i: integer;
begin
   for i := 0 to length(bytesToSend) - 1 do
      try
            FCommPort.SendByte(bytesToSend[i]);
         except
         begin
            result := LASER_NOT_CONNECTED;
            exit;
         end;
      end;
   result := TErrorType.NO_ERROR;
end;

procedure TRflExecutor.ReceiveData(var ReceivedBytes: TbyteStream; const size: byte);
var
   pdata: pWideChar;
   ndata: byte;
   i: integer;
begin
   try
      getMem(pdata, size);
      ndata := FCommPort.ReadData(pdata, size);

      SetLength(ReceivedBytes, ndata);
      for i := 0 to ndata - 1 do
         ReceivedBytes[i] := (pbyte(pdata) + i)^;

      FCommPort.FlushBuffers(True, True);
   finally
      freeMem(pdata);
   end;
end;
```

Implementazione in Delphi dei metodi SendData e ReceiveData

CONTROLLER

Il controller incaricato di coordinare i vari elementi della libreria nello svolgimento delle richieste effettuate dall'utente al dispositivo viene definito all'interno del package *controller*.

All'interno di questo file viene implementata la classe TRflController, utilizzata come interfaccia tra l'user e l'executor. Si è scelto di separare il controller dall'executor per poter permettere in caso di successive necessità di implementare dei controller con logiche di funzionalità diverse da quelle implementate dal presente senza intaccare la logica di scambio di messaggi con la seriale implementata con l'executor.

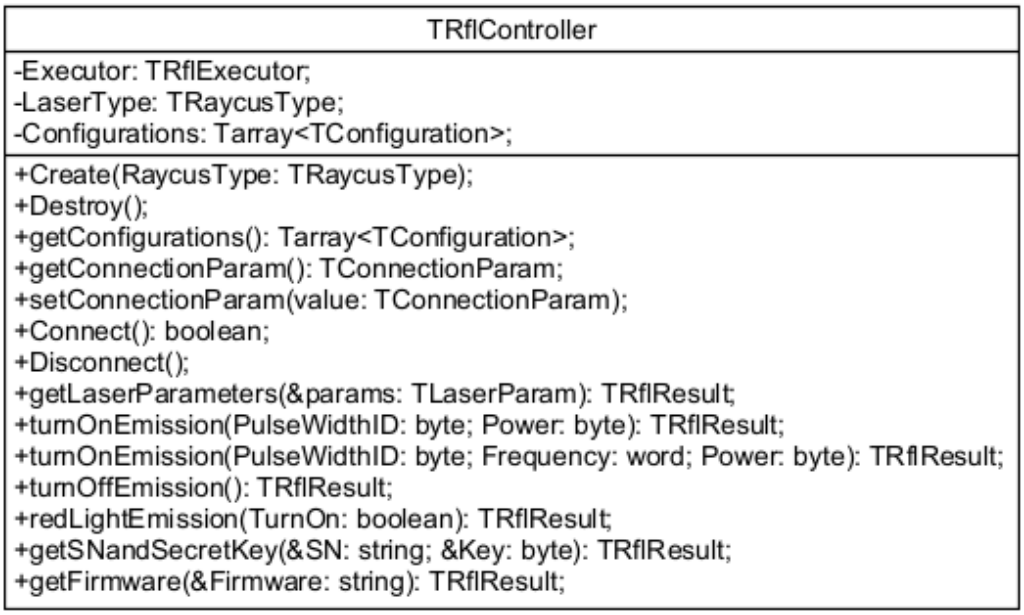


Diagramma UML di classe della classe TRflController

All'interno di ogni sua istanza vengono incapsulate un'istanza di TRflExecutor, un TRaycusType assegnato dall'utente alla creazione dell'istanza e un'array di TConfiguration selezionato in creazione dell'istanza in base al TRaycusType dato dall'utente.

La classe è dotata dei metodi per impostare la connessione dell'executor con la seriale e i metodi per impartire al laser i vari comandi disponibili.

Ogni metodo relativo ai comandi è implemento con la seguente logica:

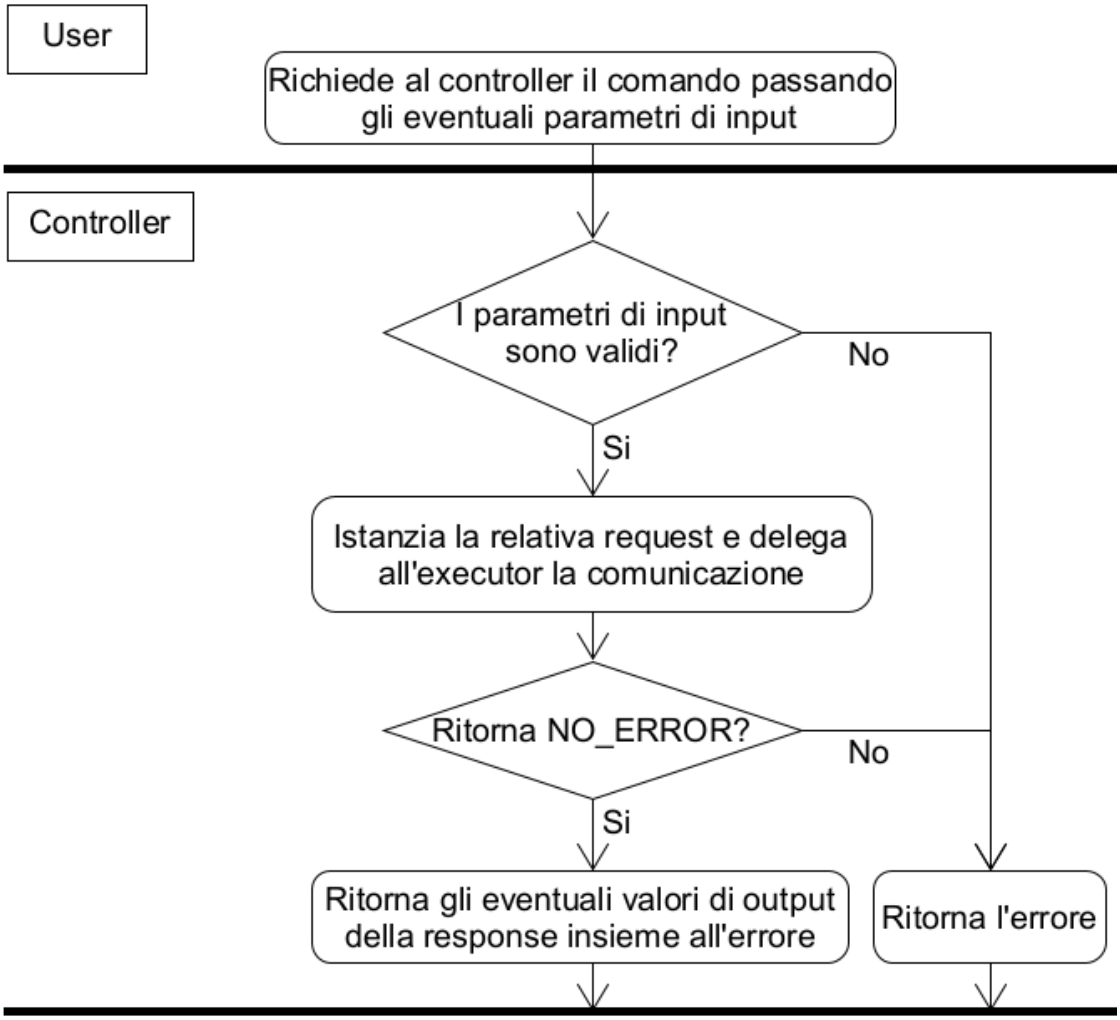


Diagramma UML di attività relativo ad una richiesta generica al dispositivo

Il controller prima istanzia la request relativa al comando richiesto impostandole gli eventuali parametri di input dati dall'utente e chiama il metodo execute dell'executor per comunicare il comando al laser, se l'output del metodo execute è un TErrorType diverso da NO_ERROR allora ritorna all'utente solo un TRflResult con valori result = false, code uguale al byte associato al TErrorType ricevuto e resultDescr uguale alla stringa di errore associata al TErrorType ricevuto, altrimenti ritorna un TRflResult con valori result = true, code = 0 e resultDescr uguale alla stringa associata al TErrorType NO_ERROR e gli eventuali valori di output del comando impartito dall'utente.

```
function TRflController.turnOnEmission(const PulseWidthID: byte; const Frequency: word; const Power: byte): TRflResult;
var
   input: TBasePacket;
   output: TBaseResponsePacket;
   error: TErrorType;

   i: byte;
begin
   if (Power < 0) or (Power > 100) then
   begin
      result := TRflResult.Create(false, Ord(POWER_OUT_OF_RANGE), enumToStr(POWER_OUT_OF_RANGE));
      exit;
   end;

   for i := 0 to length(FConfigurations) do
   begin
      if i = length(FConfigurations) then
      begin
         result := TRflResult.Create(false, Ord(PULSE_WIDTH_ID_OUT_OF_RANGE), enumToStr(PULSE_WIDTH_ID_OUT_OF_RANGE));
         exit;
      end;

      if PulseWidthID = FConfigurations[i].FPulseWidthID then
      begin
         if (Frequency < FConfigurations[i].FMinFrequency) or (Frequency > FConfigurations[i].FMaxFrequency) then
         begin
            result := TRflResult.Create(false, Ord(FREQUENCY_OUT_OF_RANGE), enumToStr(FREQUENCY_OUT_OF_RANGE));
            exit;
         end;
         break;
      end;
   end;

   try
      output := TSetParamResponse.Create();
      input := TSetParamRequest.Create();
      TSetParamRequest(input).setEmission(true);
      TSetParamRequest(input).setPulseWidthID(PulseWidthID);
      TSetParamRequest(input).setFrequency(Frequency);
      TSetParamRequest(input).setPower(Power);

      error := FExecutor.execute<TSetParamMarshaller>(input, output);
      result := TRflResult.Create(error = NO_ERROR, Ord(error), enumToStr(error));
   finally
      input.Destroy;
      output.Destroy;
   end;
end;
```

Implementazione di esempio in Delphi di un comando da richiedere al dispositivo

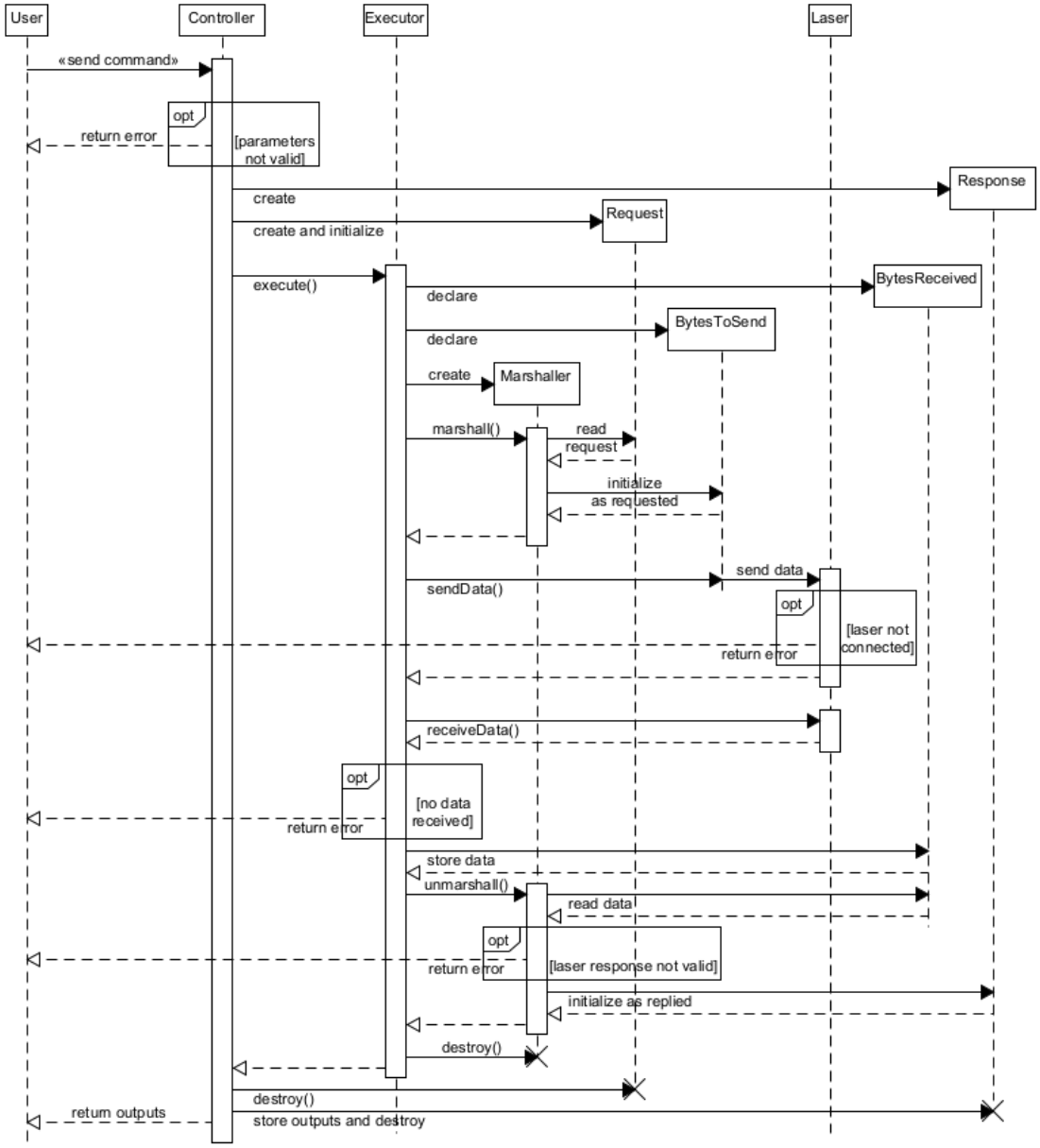


Diagramma UML di sequenza della gestione generica di un comando da parte delle varie componenti della libreria

DIAGNOSTICA DEL DISPOSITIVO

Per facilitare il controllo periodico dei parametri del dispositivo è stato creato un package *RaycusControllerUnit* esterno alla libreria che si occupa di richiedere periodicamente al dispositivo i parametri per la lettura da parte dell'utente.

All'interno di questo file vengono definite tre classi:

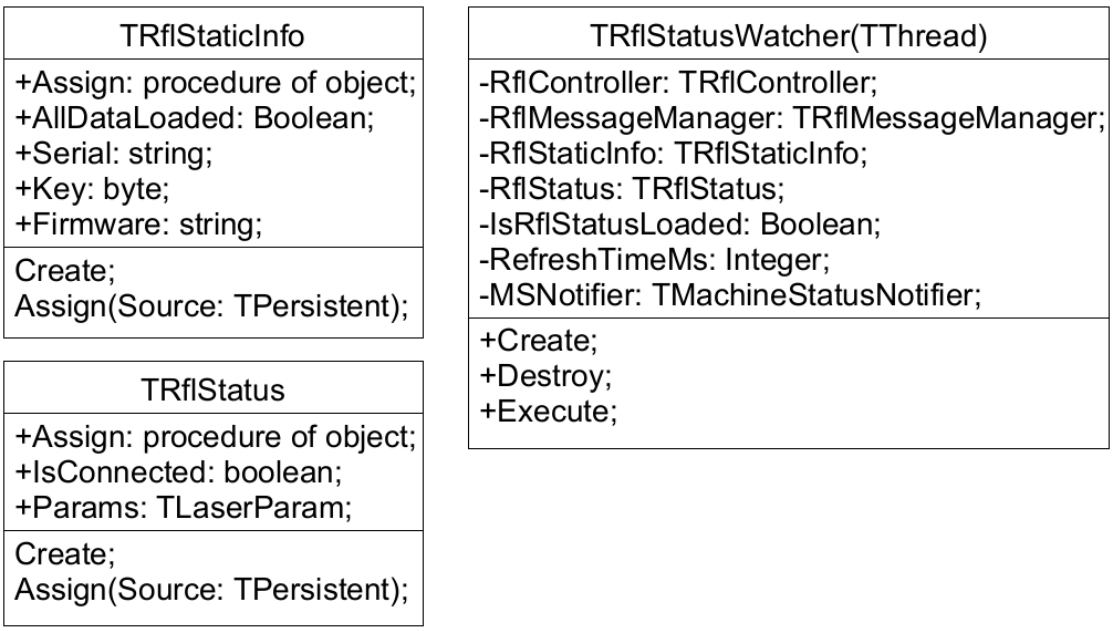
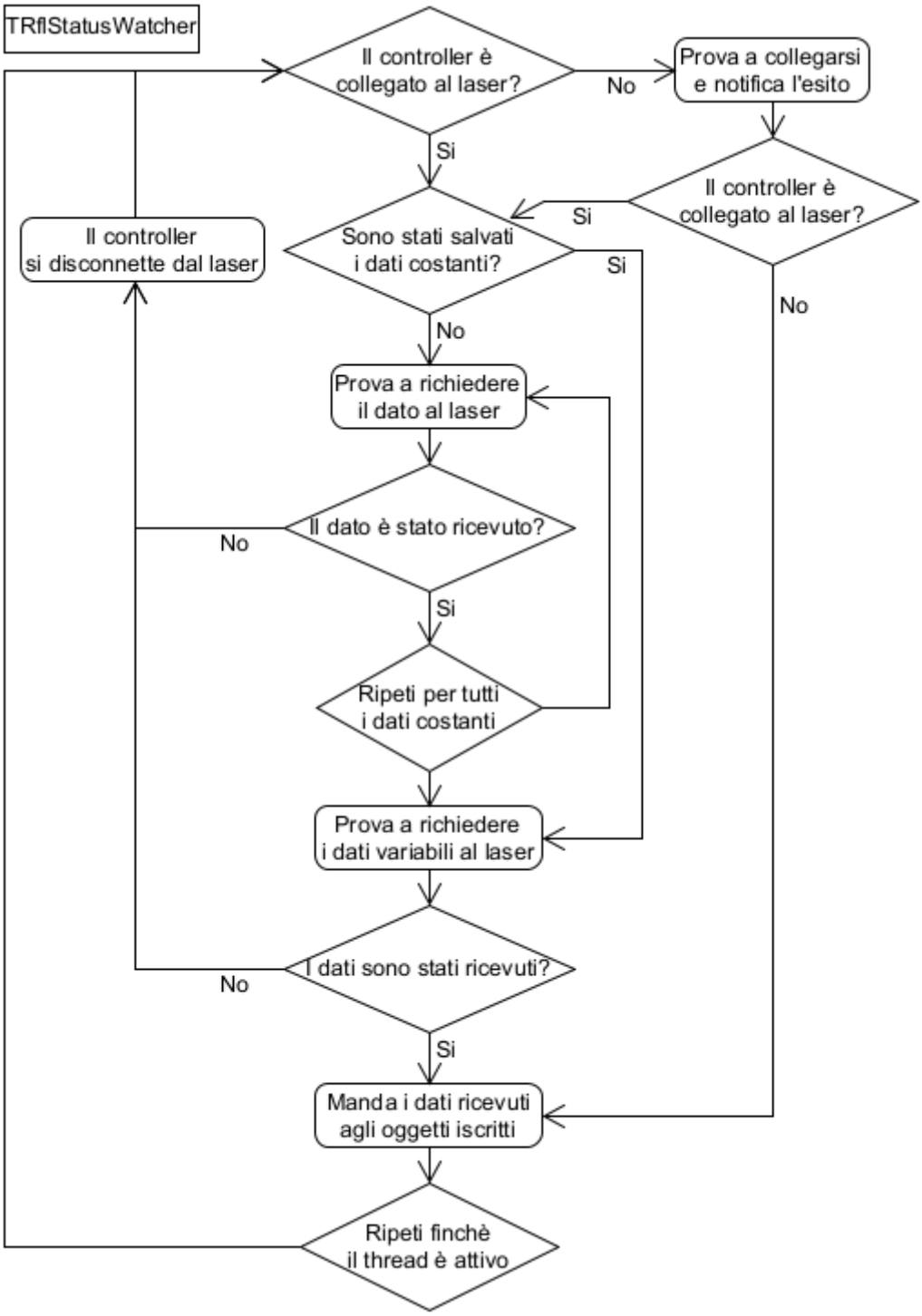


Diagramma UML di classe delle classi definite in RaycusControllerUnit

- TRflStaticInfo: contenitore in cui vengono salvate tutte le informazioni costanti del laser e che quindi richiedono una sola lettura
- TRflStatus: contenitore in cui vengono salvate tutte le informazioni del laser variabili nel tempo di cui si vuole avere un aggiornamento periodico
- TRflStatusWatcher: classe ereditata da TThread, implementa il thread di diagnostica periodica del laser rappresentato dal metodo Execute della classe e incapsula i vari oggetti di cui il thread ha bisogno per il corretto funzionamento

Inoltre offre anche i metodi GetRflControllerInstance() e GetRflMessageManagerInstance() che creano al momento la relativa istanza desiderata se non è già stata creata una volta, altrimenti restituisce un riferimento all'istanza già creata seguendo il design pattern Singleton.

Il metodo Execute della classe TRflStatusWatcher segue la seguente logica:



Il thread ripete periodicamente finchè è attivo le seguenti attività:

All'inizio verifica se il controller è collegato al laser, se non è già collegato tenta il collegamento e notifica tramite il relativo messenger dell'applicativo l'esito.

Se il collegamento non va a buon fine manda tramite i messenger relativi i dati che sono stati ricevuti fin'ora compreso l'esito negativo della tentata connessione e dopo aver aspettato un tempo di refresh deciso all'interno dell'applicativo dall'utente ritenta la connessione.

Se il controller è collegato al laser si procede con le richieste dei dati.
Se non sono già stati ottenuti tutti i dati statici al laser prima si richiede il serial number e la secret key col comando relativo, se la richiesta va a buon fine si richiede anche il firmware col relativo comando mentre se uno dei due comandi non va a buon fine si disconnette il controller dal laser e si ritorna all'inizio del ciclo dopo aver aspettato un tempo di riconnessione deciso anch'esso all'interno dell'applicativo dall'utente.

Una volta ottenute tutte le informazioni costanti si richiedono al laser le informazioni variabili tramite il comando getLaserParameters(), se la richiesta non va a buon fine si disconnette il controller dal laser e si ritorna all'inizio del ciclo dopo aver aspettato il tempo di riconnessione, altrimenti si notifica all'applicativo se e quali errori sono stati notificati dal laser.

Alla fine manda tramite i messenger relativi i dati e dopo aver aspettato il tempo di refresh ripete il ciclo.

```
if Not self.FRflStatus.IsConnected then
begin
   // mi collego
   try
      tempConnected := FRflController.connect();
   finally
      rflConnected := tempConnected;
   end;
   Self.FMSNotifier.UpdateLaserError(Self, not rflConnected, LCU_LASER_NOT_CONNECTED_ERROR);
   if rflConnected then
      self.FRflStatus.IsConnected := True
   else
      self.FRflStatus.IsConnected := False;

end;

if (self.FRflStatus.IsConnected) then
begin

   // qui recupero i dati statici; una volta caricati tutti, non li carico piu'
   if (not self.FIsRflStatusLoaded) then
   begin
      doNext := True;

      // Recupero il Serial Number e Key
      try
         tempResult := FRflController.getSNandSecretKey(sn,key);
      finally
         rflResult.Assign(tempResult);
         tempResult.Destroy;
      end;
      if rflResult.Result then
      begin
         self.FRflStaticInfo.Serial := sn;
         self.FRflStaticInfo.Key := key;
      end
      else
         doNext := False;

      if reconnectionFunct(rflResult) then continue;
      if not DoNext then continue;
```

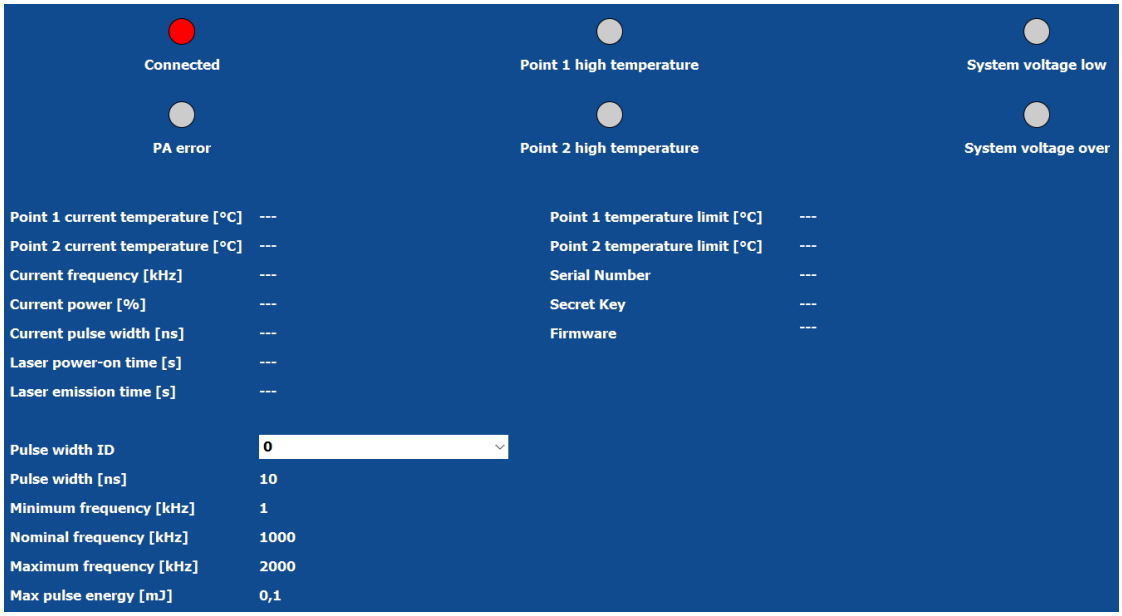
Implementazione in Delphi della parte iniziale del ciclo del thread

INTERFACCIA UTENTE

L’applicativo è stato aggiornato con due nuovi form per la gestione del dispositivo.

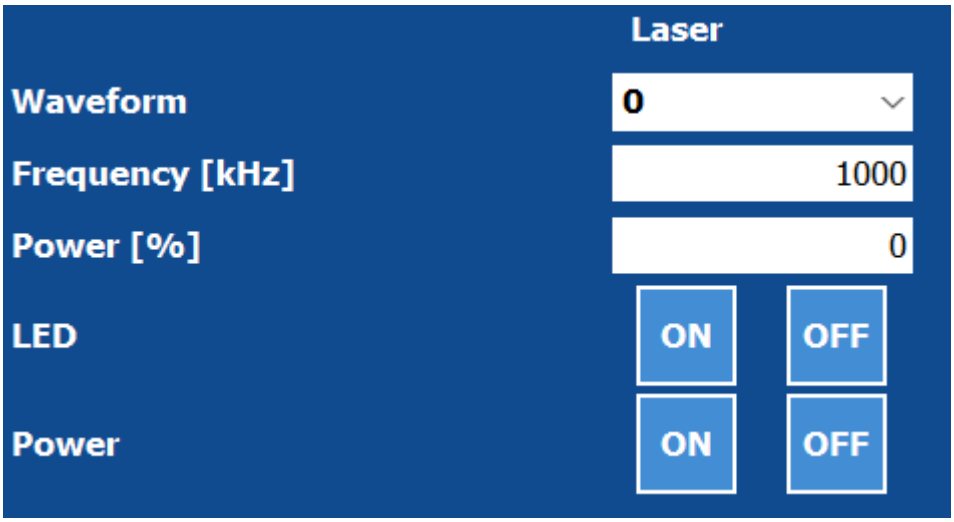
Tramite il RaycusStatusForm si può visualizzare all'interno dell'applicativo se il laser è collegato o meno, gli eventuali errori che riporta e i valori costanti e variabili che il laser manda all'applicativo tramite la ricezione dei messaggi che vengono mandati dal TRflStatusWatcher, che triggerano il metodo UpdateUI del form, incaricato di aggiornare gli elementi grafici di conseguenza, ogni volta che vengono ricevuti.

Inoltre è presente anche la tabella riportante le varie configurazioni del laser disponibili selezionando la relativa pulse width, caricata alla creazione del form in base al modello di Raycus selezionato dall'utente nelle impostazioni dell'applicativo.



Interfaccia del RaycusStatusForm

Tramite il ManualRaycusCommandForm è possibile invece richiedere l'emissione del led rosso e dell'onda laser dopo aver selezionato i parametri desiderati.



Interfaccia del ManualRaycusCommandForm

Dopo che i valori delle configurazioni vengono salvati in un' array alla creazione del form, la comboBox relativa alle waveform viene caricata di conseguenza e l'utente è obbligato a selezionare valori di frequenza compresi tra il minimo e il massimo dato dalla waveform desiderata tramite un metodo UpdateUI che riporta all'estremo più vicino il valore presente nella casella di selezione se è fuori dal suo dominio, lo stesso vale per power che deve essere compreso tra 0 e 100.

I pulsanti ON/OFF invece chiamano il rispettivo metodo del RflController e mostrano una finestra di dialogo riportante l'esito della richiesta.

```
procedure TManualRaycusCommandForm.LaserONBClick(Sender: TObject);
var
   res: TRflResult;
begin
   try
      res := getRflControllerInstance.turnOnEmission(FPulseWidthID,FFrequency,FPower);
      if res.Result then
         TDlgBoxGenericUtils.showDialog(
            Vcl.Forms.Application,
            TDlgBoxGeneric.DlgType.ok,
            LASER_COMMAND_TITLE + 'Laser ON',
            LASER_COMMAND_RESULT_OK
            )
      else
         TDlgBoxGenericUtils.showDialog(
            Vcl.Forms.Application,
            TDlgBoxGeneric.DlgType.error,
            LASER_COMMAND_TITLE + 'Laser ON',
            LASER_COMMAND_RESULT_KO + res.ResultDescr,
            );
   finally
      res.Destroy;
   end;
end;
```

I*mplementazione di esempio del metodo di accensione del laser*