Istituto Nazionale di Astrofisica
Osservatorio Astronomico di Cagliari
Località Poggio dei Pini, Strada 54, 09012 Capoterra (CA) - Italy

RECEIVER LIBRARY

Hardware + Protocol + LNAs and Dewar control library

Autore: **Marco Buttu**
Email: mbuttu@oa-cagliari.inaf.it
Versione 1.0.1, 11 novembre 2011

Receiver Library: Hardware, Protocol, LNAs and Dewar control

Marco Buttu <mbuttu@oa-cagliari.inaf.it>

Questo documento descrive come avviene il controllo dei ricevitori mediante le librerie `MicroControllerBoard` e `ReceiverControl`. Tali librerie possono essere utilizzate solo per controllare i ricevitori nei queli sono montate le schede a microcontrollore sviluppate nella Stazione Radioastronomica di Medicina (BO); Questi ricevitori ospiteranno due schede: una per il **controllo del dewar** e una per il **controllo degli LNA**.

Il protocollo di comunicazione con le schede a microcontrollore è descritto nel documento interno IRA n.358/04 (F.Fiocchi, G.Macaferri, A.Oralti, M.Morsiani). Il firmware installato nelle schede è stato realizzato da Franco Fiocchi mentre per i dettagli sull'hardware ci si può rivolgere a Alessandro Cattani ed Andrea Maccaferri.

Le librerie `MicroControllerBoard` e `ReceiverControl`, che permettono di comunicare con le schede sono state realizzate da Marco Buttu ed Andrea Orlati, e si dispongono su due livelli indipendenti: un primo livello (`MicroControllerBoard`) che fornisce un'interfaccia per la comunicazione con le schede (sostanziamente un'implementazione del protocollo di comunicazione), ed un secondo e più alto livello (`ReceiverControl`) che mediante l'utilizzo della prima libreria definisce una interfaccia per la comunicazione con il ricevitore.

Contatti

Per quanto riguarda la parte **software**:
Marco Buttu <mbuttu@oa-cagliari.inaf.it>
Andrea Orlati <a.orlati@ira.inaf.it>

Per il **firmware**:
Franco Fiocchi <f.fiocchi@ira.inaf.it>

Per quanto riguarda la parte **hardware**:
Alessandro Cattani <a.cattani@ira.inaf.it>
Andrea Maccaferri <a.maccaferri@ira.inaf.it>

Indice

Capitolo 1

Micro Controller Board Library

1.1 Introduzione

La libreria `MicroControllerBoard` consente di comunicare con le schede a micro-controllore sviluppate nella stazione radioastronomica di Medicina (Bologna)[1].

In questo capitolo non ci occuperemo esclusivamente della libreria dal punto di vista software, ma tratteremo anche aspetti di più basso livello e nascosti dalla libreria, come il protocollo di comunicazione e l'architettura hardware delle schede a microcontrollore. Lo scopo di questo documento è quindi anche quello di poter essere utilizzato come riferimento per quanto concerne la comprensione dell'implementazione software del protocollo.

1.2 Protocollo di Comunicazione

Per meglio comprendere le caratteristiche e la codifica del protocollo occorre chiarire la distinzione tra un dispositivo definito *master* ed uno *slave*, attraverso una definizione univoca:

- *master*: dispositivo che invia ad un altro dispositivo un pacchetto di dati sul canale di comunicazione;
- *slave*: dispositivo che riconosce il pacchetto di dati a lui indirizzato e, se richiesto, manda al dispositivo chiamante un pacchetto di dati in risposta.

[1] Il protocollo di comunicazione è descritto in dettaglio nel documento interno IRA n.358/04 (F.Fiocchi, G.Macaferri, A.Oralti, M.Morsiani). Il firmware che controlla le schede è stato realizzato da Franco Fiocchi mentre dell'hardware se ne sono occupati Alessandro Cattani ed Andrea Maccaferri.

Il protocollo permette la realizzazione di reti ad architettura semplice, come quella del tipo *master-slave*, ed in questo caso vi saranno dei campi di controllo ridondanti, oppure complessa come quella *multi-master*, dove dispositivi diversi possono condividere le medesime risorse.

La trasmissione di pacchetti consistenti di dati posono mettere in difficoltà i dispositivi di elaborazione dotati di micro-controllori, in quanto normalmente dotati di ristretti buffer di comunicazione. Nel protocollo è stato quindi previsto un meccanismo per frammentare la comunicazione in pacchetti di dati più piccoli denominati *frames*.

I comandi e le risposte sono suddivisi in due gruppi principali:

- *comandi estesi*: per i quali, come si evince dal loro nome, la codifica è più lunga ed è presente un campo *checksum*, utilizzato per controllare che la comunicazione tra due dispositivi sia esente da errori di trasmissione[2].

- *comandi abbreviati*: per i quali non è presente il controllo del *checksum* e nemmeno il carattere terminatore.

Entrambi i gruppi contengono i medesimi comandi (contraddistinti con identificativi diversi che ne determinano l'appartentenza), elencati di seguito:

- `CMD_INQUIRY`: chiede allo slave di restituire l'identificativo e l'esito dell'ultimo comando eseguito, insieme alla data e all'ora di esecuzione. Può essere utilizzato, per esempio, dopo un comando indirizzato a tutti gli slave che non preveda la risopsta, al fine di conoscere l'esito di uno slave specifico. Se come indicativo dell'ultimo comando eseguito, il dispositivo torna zero (0x00), significa che non ha ancora esegito alcuna operazione dal momento dell'accensione. L'esecuzione di un comando INQUIRY non viene memorizzata dai dispositivi, quindi `CMD_INQUIRY` non compare mai come identificativo dell'ultimo comando eseguito.

- `CMD_RESET`: forza lo slave a terminare ogni attività che ha in coda d'esecuzione ed a ricominiciare da capo l'esecuzione del suo programma applicativo; questo equivale a riportare il dispositivo nella stessa condizione iniziale, quando gli viene applicata l'alimentazione.

- `CMD_VERSION`: chiede allo slave di restituire il codice identificativo. Il codice restituito dipende dall'hardware del dispositivo slave e dall'applicativo su di esso sviluppato. In generale è composto da tre sezioni

[2] Il carattere di checksum si ottiene eseguendo la funzione logica EX-OR (OR esclusivo) di tutti i suoi precedenti byte incluso quello di apertura della trasmissione.

riportanti un identificativo della scheda (primi quattro caratteri), la versione del firmware (i due caratteri seguenti) e la relativa revisione (ultimi due caratteri).

- `CMD_SAVE`: permette di salvare su memoria non volatile i parametri di configurazione dello slave; questi dipendono dall'hardware del dispositivo e dall'applicativo su di esso sviluppato. I dati di configurazione sono le impostazioni di default, ovvero quelle caricate al momento dell'alimentazione della scheda.

- `CMD_RESTORE`: permette di recuperare dalla memoria non volatile i parametri di configurazione dello slave; questi dipendono dall'hardware del dispositivo e dall'applicativo su di esso sviluppato. Dopo il recupero della configurazione, il dispositivo slave esegue un reset a caldo in modo da rendere operativa la nuova configurazione, che utilizzerà tutte le volte che verrà alimentato.

- `CMD_GET_ADDR`: chiede allo slave di restituire il suo indirizzo di protocollo.

- `CMD_SET_ADDR`: comunica il nuovo indirizzo di protocollo che lo slave deve utilizzare; la riposta al comando viene data col vecchio indirizzo, dopodichè il dispositivo riposnderà solo ai comandi inviati al nuovo indirizzo. L'impostazione viene persa se viene a mancare l'alimentazione allo slave; per salvarla in modo non volatile è necessario inviare allo slave (al suo nuovo indirizzo) un comando `CMD_SAVE`.

- `CMD_GET_TIME`: chiede allo slave di restituire la data e l'orario del suo orologio interno nella seguente sequenza: secolo, anno, mese, giorno, ora, minuti, secondi e centesimi di secondo.

- `CMD_SET_TIME`: chiede allo slave di impostare l'orario e la data del suo orologio interno nella seguente sequenza: secolo, anno, mese, giorno, ora, minuti, secondi e centesimi di secondo.

- `CMD_GET_FRAME`: chiede allo slave di restituire la dimensione massima del campo PARAMETER (per i comandi) e DATA (per le risposte); questa dimensione dipende dall'hardware del dispositivo e dall'applicativo su di esso sviluppato.

- `CMD_SET_FRAME`: chiede allo slave di limitare la dimensione massima del campo PARAMETER (per i comandi) e DATA (per le risposte); questa dimensione dipende dall'hardware del dispositivo e dall'applicativo su di esso sviluppato.

- `CMD_GET_PORT`: chiede allo slave di restituire l'impostazione della porta, ovvero i parametri di configurazione necessari a definire il funzionamento; una porta parallela, ad esempio, necessita la configurazione della direzione (ingresso/uscita) dei vari bit che costituiscono la parola.

- `CMD_SET_PORT`: chiede allo slave di modificare l'impostazione della porta, ovvero i parametri di configurazione necessari a definire il funzionamento; una porta parallela, ad esempio, necessita la configurazione della direzione (ingresso/uscita) dei vari bit che costituiscono la parola.

- `CMD_GET_DATA`: chiede allo slave di restituire i dati presenti sulla porta; per una porta parallela, ad esempio, restituisce lo stato dei vari bit che costituiscono la parola.

- `CMD_SET_DATA`: chiede allo slave di modificare i dati presenti sulla porta; per una porta parallela, ad esempio, imposta lo stato dei vari bit che costituiscono la parola.

È possibile eseguire tutti i comandi appena elencati utilizzando la libreria `MicroControllerBoard`, la quale si occupa anche della verifica della risposta ed eventualmente del controllo degli errori mediante checksum.

Per maggiori dettagli sul protocollo si veda [4].

1.3 Il controllo del Dewar e degli LNA

Come già detto in precedenza, nella stazione radioastronomica di Medicina (BO) sono state progettate e realizzate delle scheda a microcontrollore con le quali è possibile dialogare utilizzando il protocollo descritto nella sezione 1.2. Su ciascun ricevitore vengono installate due di queste schede, una utilizzata per il controllo del *dewar* e l'altra per il controllo degli LNA. Nelle sezioni che seguono è descritta in dettaglio l'architettura di queste due schede.

1.3.1 LNAs control board

La scheda ha una duplice funzione: alimentare gli LNA (Low Noise Amplifier) dei feed e permettere la lettura dei valori delle grandezze V_D, I_D e V_G di ogni *stadio* degli LNA di ciascun feed, e per ogni *canale*. La scheda è divisa in due sezioni, una per la *polarizzazione* left e l'altra per la right (due canali), ed ogni sezione può alimentare 5 LNA (5 stadi di amplificazione).

Le letture vengono fatte sulla porta AD24, ed ogni lettura consente di recuperare i valori delle grandezze di 8 canali per volta (4 feed, 2 canali per

feed). Per poter effettuare una lettura è necessario indicare quali grandezze andare a leggere, e questo viene fatto andando a scrivere sulla porta DIO.

Le porte di nostro interesse quindi sono due:

- DIO: è una porta a 16 bit accessibile in lettura/scrittura;
- AD24: è una porta con 8 locazioni da 32 bit (4 byte per ogni locazione).

Iniziamo con la descrizione della porta AD24, schematizzata in figura 1.1. Una lettura dalla porta AD24 fornisce quindi un dato composto da 32 byte (ogni locazione $AD8, \ldots, AD15$ è composta da 4 byte). Dalla locazione AD8 sarà possibile leggere la grandezza di interesse per il canale left di uno dei seguenti feed: 0, 1, 8 o 9, mentre dalla locazione AD9 sarà possibile leggere il valore del canale right per gli stessi feed, e così via per tutti gli altri feed sulla base dello schema di figura 1.1.

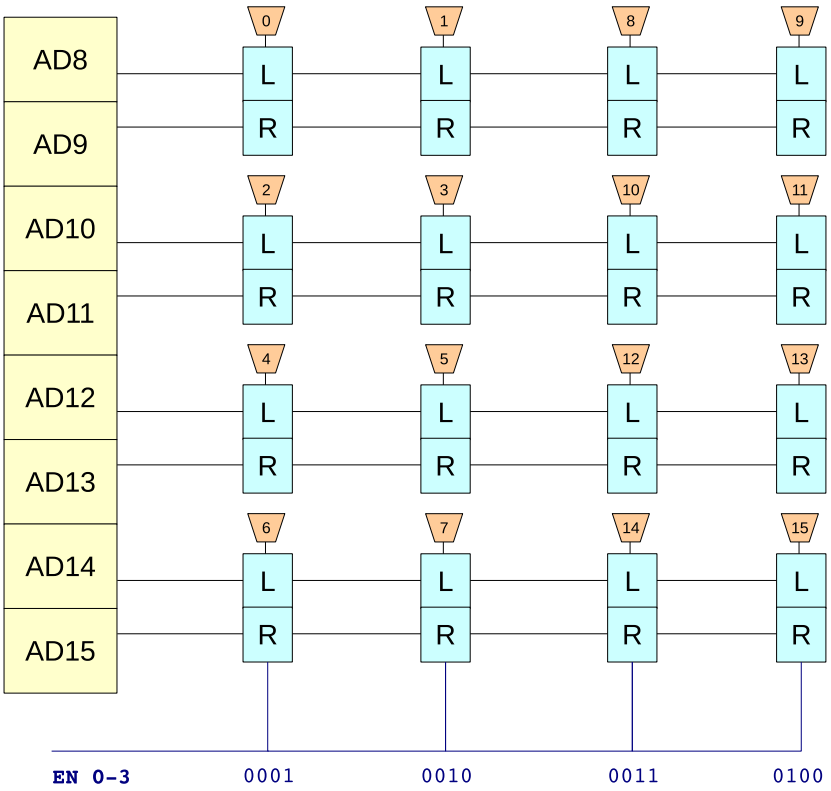

Figura 1.1: Schema a blocchi della porta AD24

È possibile selezionare i feed su cui andare a leggere impostando sulla porta DIO il valore di `EN 0-3`; come si vede in figura il valore `0001` di `EN 0-3`

permette di selezionare la prima colonna, il valore `0010` la seconda e così via per le altre due. La grandezza da leggere invece dipende dal valore `AD 0-3` impostato nel DIO; questo permette di selezionare il valore di V_D, I_D o V_G di uno dei cinque stadi degli LNA. La porta DIO è schematizzata in figura 1.2.

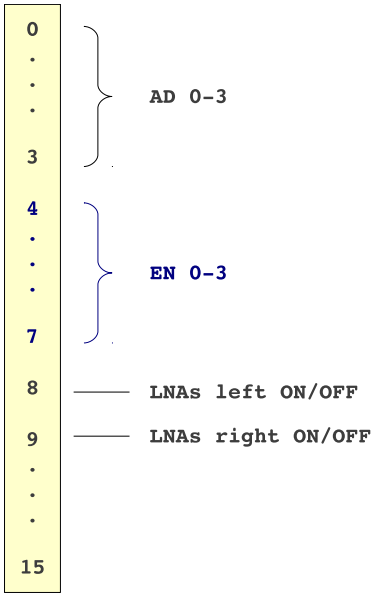

Figura 1.2: Schema della porta DIO

Nella tabella 1.1 sono indicati i valori di `AD 03` da impostare sulla porta DIO per selezionare la grandezza da leggere.

`AD 0-3`	**Segnale**	**Stadio**
`0000`	V_D	primo
`0001`	I_D	primo
`0010`	V_G	primo
`0011`	V_D	secondo
...	...	...

Tabella 1.1: Valori di `AD 0-3` per la selezione della grandezza da leggere

Supponiamo ad esempio di voler leggere gli 8 valori (4 feed, 2 canali per feed) di V_G del terzo stadio dei feed 8, 10, 12 e 14. Il valore di `EN 0-3` sarà `0011` mentre il valore di `AD 0-3` sarà `1000`. Faremo allora due richieste: la prima (`SET_DATA`) ha lo scopo di configurare la porta DIO in modo da

impostare i valori di `AD 0-3` e `EN 0-3`, mentre la seconda (`GET_DATA`) è la lettura della grandezza richiesta (V_G del terzo stadio per i vari feed):

- `SET_DATA`: avrà quattro *parametri*:
  1. **data type**: unsigned da 8 bit
  2. **port type**: DIO
  3. **port number**: da 0 a 7 (8 bit, dall'indice 0 all'indice 7)
  4. **value**: `10000011` (`AD 0-3` + `EN 0-3`)
- `GET_DATA`: avrà tre parametri:
  1. **data type**: 32 bit floating point
  2. **port type**: AD24
  3. **port number**: da 8 a 15

Dopo aver dato il comando `SET_DATA` è necessario attendere un *tempo di guardia* prima di andare a leggere i valori con `GET_DATA`, in modo da avere le uscite stabili; questo tempo di guardia non deve essere inferiore ai 200 ms.

Il dato letto con il `GET_DATA` è un 32 byte, che una volta scomposto in elementi da 4 byte ci fornisce 8 valori analogici di tensione, che andranno poi convertiti secondo opportune formule di trasformazione.

1.3.2 Dewar Control Board

Anche la scheda per il controllo del *dewar* ha una porta DIO ed una porta AD24, ma a differenza della scheda per il controllo degli LNA non è necessario scrivere sulla porta DIO prima di leggere dalla porta AD24, poichè non è possibile specificare quali grandezze leggere dalla porta AD24; queste infatti non cambiano e sono le seguenti:

1. `AD08`: temperatura criogenica numero 1;
2. `AD09`: temperatura criogenica numero 2;
3. `AD10`: pressione all'interno del dewar;
4. `AD11`: temperatura criogenica numero 3;
5. `AD12`: temperatura criogenica numero 4;
6. `AD13`: spare temperature;

7. `AD14`: vertex temperaturetemperatura!vertex;

8. `AD15`: spare temperature.

I valori di tensione letti andranno poi convertiti tramite opportune formule in modo da ottenere i valori nelle corrette unità ingegneristiche (ad esempio in Kelvin o mbar).

Le porte ed i *numeri di porta* descritti in questo documento sono relativi alle schede montate sul ricevitore 22 GHz multi-beam in funzione su SRT, ma le schede che verranno montate sui futuri ricevitori non dovrebbero cambiare questa mappatura. I dettagli tecnici sulla scheda per il controllo degli LNA e su quella per il controllo del dewar sono reperibili rispettivamente nei documenti [2] e [3].

1.4 Utilizzo della Libreria

La `MicroControllerBoard` è una libreria *thread-safe* composta da 3 file: *MicroControllerBoard.h* e *MicroControllerBoard.cpp* nei quali viene rispettivamente dichiarata e definita la classe `MicroControllerBoard`, ed il file *MicroControllerBoardDef.h* nel quale sono definiti tutti i parametri, come i *tipi di dato*, le porte ed i numeri di porta.

Nel listato 1.1 è illustrato un breve esempio di utilizzo della libreria, nel quale viene istanziato un oggetto MicroControllerBoard al fine di effettuare una richiesta `GET_DATA`.

```
1  #include "MicroControllerBoard.h"
2  #include <cstdlib>
3  using namespace IRA;
4
5  int main()
6  {
7      // Definiamo i parametri IP e porta da passare al costruttore
8      std::string IP("192.168.51.63"); // Indirizzo IP della scheda
9      unsigned int port = 5002; // Numero di porta della connessione
10     // Il vettore params conterra' i parametri del GET_DATA
11     std::vector<BYTE> params;
12     // Il data type e' un floating point a 32
13     params.push_back(MCB_CMD_DATA_TYPE_F32);
14     // La porta dalla quale leggere i dati e' la AD24
15     params.push_back(MCB_PORT_TYPE_AD24);
16     // Vogliamo leggere un insieme di 8 dati floating point
17     params.push_back(MCB_PORT_NUMBER_00_07);
```

```
18      try {
19          MicroControllerBoard mcb = MicroControllerBoard(IP, port);
20          // Creiamo la connessione TCP/IP con la scheda
21          mcb.openConnection();
22          // Effettuiamo una richiesta GET_DATA
23          mcb.send(MCB_CMD_GET_DATA, params);
24          data = mcb.receive(); // Riceviamo i dati richiesti
25          // Richiesta GET_DATA senza controllo checksum
26          mcb.send(MCB_CMD_GET_DATA | MCB_CMD_TYPE_NOCHECKSUM, params);
27          data = mcb.receive();
28          mcb.closeConnection(); // Chiudiamo la connessione
29      }
30      catch(MicroControllerBoardEx& ex) {
31          cout << ex.what() << endl;
32          mcb.closeConnection();
33          // ...
34      }
35      return 0;
36  }
```

Listato 1.1: Esempio di utilizzo della libreria `MicroControllerBoard`

Alle righe 11-17 viene creato il vettore di parametri che caratterizza la richiesta. I codici del *data type*, *port type* e *port number* sono definiti nel file MicroControllerBoardDef.h, così come i codici dei comandi che vengono passati al metodo `send` alle righe 23 e 26.

Quando nel metodo `send` il codice di un comando viene messo in `or` con `MCB_CMD_TYPE_NOCHECKSUM` (ad esempio, riga 26), significa che si sta effettuando una comunicazione priva di controllo degli errori (nessun checksum).

Alle righe 24 e 27 viene invocato il metodo `receive`, il quale effettua tutti i possibili controlli sulla validità della risposta:

- gli indirizzi del master e dello slave della risposta devono coincidere con quelli della richiesta;
- il *codice del comando* deve essere valido;
- risposta e richiesta devono avere lo stesso codice del comando;
- l'identificativo (ID) della risposta deve essere lo stesso della richiesta;
- il numero di parametri deve essere corretto;
- il *checksum* (se richiesto) deve essere corretto.

Capitolo 2

Receiver Control Library

2.1 Introduzione

La `ReceiverControl` library definisce una interfaccia di alto livello tramite la quale comunicare con il ricevitore. Poichè, come è stato detto, su ogni ricevitore vengono installate due schede (una per il controllo del dewar ed un'altra per alimentare gli LNA e leggerne i valori di V_D, I_D e V_G), un oggetto `ReceiverControl` istanzierà due `MicroControllerBoard`; questo come tutti gli altri dettagli di basso livello sarà trasparente per l'utilizzatore della classe.

La libreria inoltre è *thread-safe* in quanto per comunicare con le schede si appoggia alla `MicroControllerBoard` (come detto nel capitolo precedente, la `MicroControllerBoard` è thread-safe).

2.2 Interfaccia

L'interfaccia della classe espone una serie di metodi che consentono di avere un completo controllo del ricevitore.

2.2.1 Il Costruttore

Nel listato 2.1, mostrato di seguito, è riportata la dichiarazione del *costruttore* della classe `ReceiverControl`.

```
1 ReceiverControl(
2         const std::string dewar_ip,
3         const unsigned short dewar_port,
4         const std::string lna_ip,
5         const unsigned short lna_port,
6         const unsigned short number_of_feeds=1,
```

```
7         const BYTE dewar_madd=0x7C, // Dewar board master address
8         const BYTE dewar_sadd=0x7D, // Dewar board slave address
9         const BYTE lna_madd=0x7C, // LNA board master address
10        const BYTE lna_sadd=0x7D, // LNA board slave address
11        bool reliable_comm=true,
12        const unsigned int guard_time=250000 // 0.25 seconds
13 ) throw (ReceiverControlEx);
```

Listato 2.1: Dichiarazione del costruttore della classe `ReceiverControl`

I parametri hanno il seguente significato:

- `dewar_ip`: l'indirizzo IP della scheda per il controllo del dewar;
- `dewar_port`: la porta utilizzata nella connessione verso la scheda per il controllo del dewar;
- `lna_ip`: l'indirizzo IP della scheda per il controllo degli LNA;
- `lna_port`: la porta utilizzata nella connessione verso la scheda per il controllo degli LNA;
- `number_of_feeds`: numero di feed del ricevitore (1 di default);
- `dewar_maddr`: l'indirizzo del master della scheda del dewar (il valore di default è 0x7D);
- `dewar_saddr` l'indirizzo dello slave della scheda del dewar (il valore di default è 0x7F)
- `lna_maddr`: l'indirizzo del master della scheda degli LNA (il valore di default è 0x7D);
- `lna_saddr`: l'indirizzo dello slave della scheda degli LNA (il valore di default è 0x7F)
- `reliable_comm`: quando vale `true` allora la comunicazione con le schede avviene con controllo degli errori (checksum). Di default è `true`;
- `guard_time`: quando viene fatta una richiesta di lettura dei dati alla scheda degli LNA, dobbiamo aspettare un certo lasso di tempo prima di leggere il dato richiesto, in modo tale che le uscite si stabilizzino. Questo tempo non deve essere inferiore ai 200ms ed è impostabile tramite questo parametro.

2.2.2 Il metodo `fetValues`

Il metodo `fetValues` restituisce la struttura `FetValues` di un LNA per un dato stadio e per un dato feed. La struttura è definita di seguito:

```
struct FetValues {
    double VDL; // Drain Voltage, left channel [V]
    double IDL; // Drain Current, left channel [mA]
    double VGL; // Gate Voltage, left channel [V]
    double VDR; // Drain Voltage, right channel [V]
    double IDR; // Drain Current, right channel [mA]
    double VGR; // Gate Voltage, right channel [V]
};
```

Per poter restiture la struttura `FetValues` dobbiamo quindi passare al metodo `fetValues` i seguenti parametri:

- il codice identificativo del feed;
- il numero dello stadio;
- un puntatore alla funzione di conversione che, a partire dal valore in tensione letto, lo converte in un valore di corrente (IDL e IDR);
- un puntatore alla funzione che converte il valore letto nel corretto valore di tensione (VDL, VGL, VDR, VGR).

```
/** return the FetValues (VDL, IDL, VGR, VDR, IDR and VGR)
 *  of the LNA of the feed ‘feed_number‘ and stage ‘stage_number‘.
 *  The letter L means that the value is referred to the left
 *  channel, the R if for the right one.
 *  @param feed_number the ID code of the feed (from 0 to 15)
 *  @param stage_number the stage number (from 1 to 5)
 *  @param currentConverter pointer to the function that performs
 *  the conversion from voltage to mA; default value is NULL,
 *  and in this case the value of ID is the voltage value (the
 *  value before conversion).
 *  @param voltageConverter pointer to the function that performs
 *  the conversion from voltage to voltage; default value is NULL,
 *  and in this case the values of VD and VG are the voltage values
 *  before the conversion.
 *  @return the low cryogenic temperature in Kelvin if
 *  converter != NULL, the value in voltage (before conversion)
 *  otherwise.
```

```
 *  @return the FetValues, a struct of three double members:
 *  VD [V], ID [mA] and VG [V] after the conversion if the
 *  function pointers are not NULL, the values before conversion
 *  (voltage) if the pointers are NULL.
 *  @throw ReceiverControlEx
 */
FetValues fetValues(
        unsigned short feed_number,
        unsigned short stage_number,
        double (*currentConverter)(double voltage) = NULL,
        double (*voltageConverter)(double voltage) = NULL
) throw (ReceiverControlEx);
```

Listato 2.2: Dichiarazione del metodo `fetValues`

2.2.3 Il metodo `stageValues`

Il metodo `stageValues` prende come parametri: un elemento `FetValue` (ovvero `DRAIN_VOLTAGE`, `DRAIN_CURRENT` o `GATE_VOLTAGE`), lo stadio di amplificazione (un intero compreso tra 1 e 5), un puntatore ad una funzione che effettua la conversione del valore letto nel corrispondente valore di tensione o corrente, a seconda della grandezza richiesta (primo parametro).

Il tipo restituito è uno `StageValues`:

```
/**
 * The left_channel member stores all the left channel values of a
 * specific fet quantity (VD, ID or VG), and the right one stores
 * the values for the right channel.
 */
struct StageValues {
    std::vector<double> left_channel;
    std::vector<double> right_channel;
};
```

Il membro `left_channel` è un vettore che ha come elementi i valori della grandezza richiesta (il primo parametro del metodo), per lo stadio di amplificazione richiesto, ordinati per feed; il primo elemento del vettore è quindi riferito al primo feed, il secondo elemento al secondo feed e così via. Stessa cosa per quanto riguarda il vettore `right_channel`. Per poter recuperare

questi valori, sulla base di quanto descritto in 1.3.1, dobbiamo fare un numero di richeste `GET_DATA` che dipende solo dal numero di feed[1].

```
/** Return for each feed and channel the fet quantity value of
 *  a given stage. For instance, if you want to get the VD values
 *  of all the feeds related to the amplifier stage N, you must
 *  call the method like so: stageValues(DRAIN_VOLTAGE, N).
 *
 *  @param quantity a FetValue: DRAIN_VOLTAGE, DRAIN_CURRENT
 *  or GATE_CURRENT
 *  @param stage_number the stage number (from 1 to 5)
 *  @param converter pointer to the function that performs the
 *  conversion from voltage to the right unit or just with a
 *  scale factor; default value is NULL, and in this case the
 *  value returned is without conversion.
 *  @return the StageValues for a given fet ``quantity`` and
 *  ``stage_number``. The StageValues is a struct of two
 *  members std::vector<double>, one member for the left
 *  channel and one for the right one. That members contain
 *  the related quantities of all the feeds; that means each
 *  item of the std::vector<double> is the quantity value of
 *  a feed, for the stage requested.
 *  @throw ReceiverControlEx
 */
 StageValues stageValues(
        FetValue quantity,
        unsigned short stage_number,
        double (*converter)(double voltage)=NULL
) throw (ReceiverControlEx);
```

Listato 2.3: Dichiarazione del metodo `stageValues`

2.2.4 Il metodo `turnLeftLNAsOn`

Il metodo `turnLeftLNAsOn` accende tutti gli LNA dei canali left.

```
/** Turn the LNAs of the left channels ON
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 08
```

[1]Nel caso del ricevitore 22GHz multi-beem ad esempio, avendo questo 7 feed dovremo fare 2 richieste `GET_DATA`, poichè sono 2 le colonne da leggere, come si evince dallo schema di figura 1.1.

```
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void turnLeftLNAsOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_08,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.4: Dichiarazione del metodo `turnLeftLNAsOn`

2.2.5 Il metodo `turnLeftLNAsOff`

Il metodo `turnLeftLNAsOff` spegne tutti gli LNA dei canali left.

```
/** Turn the LNAs of the left channels OFF
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default port number is 08
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void turnLeftLNAsOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_08,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.5: Dichiarazione del metodo `turnLeftLNAsOff`

2.2.6 Il metodo `turnRightLNAsOn`

Il metodo `turnRightLNAsOn` accende tutti gli LNA dei canali right.

```
/** Turn the LNAs of the right channels ON
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 09
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
```

```
 */
void turnRightLNAsOn(
        const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
        const BYTE port_type=MCB_PORT_TYPE_DIO,
        const BYTE port_number=MCB_PORT_NUMBER_09,
        const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.6: Dichiarazione del metodo `turnRightLNAsOn`

2.2.7 Il metodo `turnRightLNAsOff`

Il metodo `turnRightLNAsOff` spegne tutti gli LNA dei canali right.

```
/** Turn the LNAs of the right channels OFF
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 09
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void turnRightLNAsOff(
        const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
        const BYTE port_type=MCB_PORT_TYPE_DIO,
        const BYTE port_number=MCB_PORT_NUMBER_09,
        const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.7: Dichiarazione del metodo `turnRightLNAsOff`

2.2.8 Il metodo `setCalibrationOn`

Il metodo `setCalibrationOn` avvia la calibrazione del ricevitore mediante una marca di rumore.

```
/** Set the noise mark generator to ON
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 11
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
```

```
void setCalibrationOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_11,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.8: Dichiarazione del metodo `setCalibrationOn`

2.2.9 Il metodo `setCalibrationOff`

Il metodo `setCalibrationOff` ferma la calibrazione del ricevitore.

```
/** Set the noise mark generator to OFF
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 11
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void setCalibrationOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_11,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.9: Dichiarazione del metodo `setCalibrationOff`

2.2.10 Il metodo `isCalibrationOn`

Il metodo `isCalibrationOn` restituisce `true` se la calibrazione del ricevitore è attiva.

```
/** Is the noise mark generator set to ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 11
 *  @return true if the noise mark generator is set to ON
 *  @throw ReceiverControlEx
 */
bool isCalibrationOn(
```

```
9         const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
10        const BYTE port_type=MCB_PORT_TYPE_DIO,
11        const BYTE port_number=MCB_PORT_NUMBER_11
12 ) throw (ReceiverControlEx);
```

Listato 2.10: Dichiarazione del metodo `isCalibrationOn`

2.2.11 Il metodo `setExtCalibrationOn`

Il metodo `setExtCalibrationOn` abilita il comando esterno per la calibrazione. ricevitore è attiva.

```
1  /** Enable the external noise mark generator synchronous command
2   *  @param data_type the type of the data; the default is 1 bit
3   *  @param port_type the port type; the default is the Digital IO
4   *  @param port_number the port number; the default is 12
5   *  @param value the value to set; the default value is 0x01
6   *  @throw ReceiverControlEx
7   */
8  void setExtCalibrationOn(
9         const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
10        const BYTE port_type=MCB_PORT_TYPE_DIO,
11        const BYTE port_number=MCB_PORT_NUMBER_12,
12        const BYTE value=0x01
13 ) throw (ReceiverControlEx);
```

Listato 2.11: Dichiarazione del metodo `setExtCalibrationOn`

2.2.12 Il metodo `setExtCalibrationOff`

Il metodo `setExtCalibrationOff` disabilita il comando esterno per fermare la calibrazione.

```
1  /** Disable the external noise mark generator synchronous command
2   *  @param data_type the type of the data; the default is 1 bit
3   *  @param port_type the port type; the default is the Digital IO
4   *  @param port_number the port number; the default is 12
5   *  @param value the value to set; the default value is 0x00
6   *  @throw ReceiverControlEx
7   */
8  void setExtCalibrationOff(
9         const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
```

```
10          const BYTE port_type=MCB_PORT_TYPE_DIO,
11          const BYTE port_number=MCB_PORT_NUMBER_12,
12          const BYTE value=0x00
13  ) throw (ReceiverControlEx);
```

Listato 2.12: Dichiarazione del metodo `setExtCalibrationOff`

2.2.13 Il metodo `isExtCalibrationOn`

Il metodo `isExtCalibrationOn` restituisce `true` se il comando esterno per avviare la calibrazione è abilitato.

```
1   /** Is the external noise mark generator command enabled?
2    *  @param data_type the type of the data; the default is 1 bit
3    *  @param port_type the port type; the default is the Digital IO
4    *  @param port_number the port number; the default is 12
5    *  @return true if the external noise mark generator command
6    *  is enabled
7    *  @throw ReceiverControlEx
8    */
9   bool isExtCalibrationOn(
10          const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
11          const BYTE port_type=MCB_PORT_TYPE_DIO,
12          const BYTE port_number=MCB_PORT_NUMBER_12
13  ) throw (ReceiverControlEx);
```

Listato 2.13: Dichiarazione del metodo `isExtCalibrationOn`

2.2.14 Il metodo `setReliableCommOn`

Il metodo `setReliableCommOn` attiva la comunicazione con controllo degli errori (checksum).

```
1   /** Set the reliable communication to/from the board to ON */
2   void setReliableCommOn() { m_reliable_comm = true; }
```

Listato 2.14: Dichiarazione del metodo `setReliableCommOn`

2.2.15 Il metodo `setReliableCommOff`

Il metodo `setReliableCommOff` disattiva la comunicazione con controllo degli errori (nessun checksum).

```
/** Set the reliable communication to/from the board to OFF */
void setReliableCommOff() { m_reliable_comm = false; }
```

Listato 2.15: Dichiarazione del metodo `setReliableCommOff`

2.2.16 Il metodo `isReliableCommOn`

```
/** return true if the communication to the board is
 *  set to be reliable
 */
bool isReliableCommOn() { return m_reliable_comm; }
```

Listato 2.16: Dichiarazione del metodo `isReliableCommOn`

2.2.17 Il metodo `vacuum`

Il metodo `vacuum` restituisce la pressione (in *mbar*) all'interno del dewar.

```
/** Return the vacuum value inside the dewar in mbar
 *
 *  @param converter pointer to the function that performs
 *  the conversion from voltage to vacumm unit [mbar];
 *  default value is NULL, and in this case the value
 *  returned by vacuum is the voltage value (the value
 *  before conversion).
 *  @param data_type the type of the data; the default
 *  type is a 32 bit floating point
 *  @param port_type the port type; the default is the AD24
 *  @param port_number the port number; the default value
 *  is a range of port numbers from 8 to 15.
 *  @param raw_index the index that allows to get the vacuum
 *  value from the port_number range.
 *  The default value is 2.
 *  @return the vacuum inside the dewar in mbar if
 *  converter != NULL, the value before conversion otherwise.
 *  @throw ReceiverControlEx
 */
double vacuum(
       double (*converter)(double voltage)=NULL,
       const BYTE data_type=MCB_CMD_DATA_TYPE_F32,
       const BYTE port_type=MCB_PORT_TYPE_AD24,
```

```
24         const BYTE port_number=MCB_PORT_NUMBER_00_07,
25         const size_t raw_index=2
26 ) throw (ReceiverControlEx);
```

Listato 2.17: Dichiarazione del metodo `vacuum`

2.2.18 Il metodo `vertexTemperature`

```
1  /** Return the vertex temperature in K
2   *  @param converter pointer to the function that
3   *  performs the conversion from voltage to Kelvin;
4   *  default value is NULL, and in this case the value
5   *  returned by vertexTemperature is the voltage value
6   *  (the value before conversion).
7   *  @param data_type the type of the data; the default
8   *  type is a 32 bit floating point
9   *  @param port_type the port type; the default is the AD24
10  *  @param port_number the port number; the default value
11  *  is a range of port numbers from 8 to 15.
12  *  @param raw_index the index that allows to get the
13  *  vertex temperature value from the port_number range.
14  *  The default value is 6.
15  *  @return the vertex temperature in Kelvin if
16  *  converter != NULL, the value before conversion otherwise.
17  *  @throw ReceiverControlEx
18  */
19 double vertexTemperature(
20         double (*converter)(double voltage)=NULL,
21         const BYTE data_type=MCB_CMD_DATA_TYPE_F32,
22         const BYTE port_type=MCB_PORT_TYPE_AD24,
23         const BYTE port_number=MCB_PORT_NUMBER_00_07,
24         const size_t raw_index=6
25 ) throw (ReceiverControlEx);
```

Listato 2.18: Dichiarazione del metodo `vertexTemperature`

2.2.19 Il metodo `cryoTemperature`

```
1  /** Return the cryogenic temperature
2   *  @param temperature_id the id code of the
```

```
 *  temperature (1, 2, 3 or 4)
 *  @param converter pointer to the function
 *  that performs the conversion from
 *  voltage to Kelvin; default value is NULL,
 *  and in this case the value returned by
 *  cryoTemperature is the voltage value (the
 *  value before conversion).
 *  @param data_type the type of the data; the
 *  default type is a 32 bit floating point
 *  @param port_type the port type; the default is the AD24
 *  @param port_number the port number; the default value
 *  is a range of port numbers from 8 to 15.
 *  @return the cryogenic temperature in Kelvin if
 *  converter != NULL, the value in voltage (before
 *  conversion) otherwise.
 *  @throw ReceiverControlEx
 */
double cryoTemperature(
       const short temperature_id,
       double (*converter)(double voltage)=NULL,
       const BYTE data_type=MCB_CMD_DATA_TYPE_F32,
       const BYTE port_type=MCB_PORT_TYPE_AD24,
       const BYTE port_number=MCB_PORT_NUMBER_00_07
) throw (ReceiverControlEx);
```

Listato 2.19: Dichiarazione del metodo `cryoTemperature`

2.2.20 Il metodo `setCoolHeadOn`

```
/** Set to ON the cool head
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 08
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void setCoolHeadOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_08,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.20: Dichiarazione del metodo `setCoolHeadOn`

2.2.21 Il metodo `setCoolHeadOff`

```
/** Set to OFF the cool head
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 08
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void setCoolHeadOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_08,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.21: Dichiarazione del metodo `setCoolHeadOff`

2.2.22 Il metodo `isCoolHeadOn`

```
/** Is the cool head ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default number is 08
 *  @return true if the cool head is ON
 *  @throw ReceiverControlEx
 */
bool isCoolHeadOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_08
) throw (ReceiverControlEx);
```

Listato 2.22: Dichiarazione del metodo `isCoolHeadOn`

2.2.23 Il metodo `setVacuumSensorOn`

Il metodo `setVacuumSensorOn` attiva il sensore di pressione.

```
/** Set to ON the vacuum sensor
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 04
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void setVacuumSensorOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_04,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.23: Dichiarazione del metodo `setVacuumSensorOn`

2.2.24 Il metodo `setVacuumSensorOff`

Il metodo `setVacuumSensorOff` disattiva il sensore di pressione.

```
/** Set to OFF the vacuum sensor
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 04
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void setVacuumSensorOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_04,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.24: Dichiarazione del metodo `setVacuumSensorOff`

2.2.25 Il metodo `isVacuumSensorOn`

Il metodo `isVacuumSensorOn` restituisce true se il sensore di pressione è attivo.

```
/** Is the vacuum sensor ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 04
 *  @return true if the vacuum sensor is ON
 *  @throw ReceiverControlEx
 */
bool isVacuumSensorOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_04
) throw (ReceiverControlEx);
```

Listato 2.25: Dichiarazione del metodo `isVacuumSensorOn`

2.2.26 Il metodo `setVacuumPumpOn`

Il metodo `setVacuumPumpOn` attiva la *pompa a vuoto*.

```
/** Set to ON the vacuum pump
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 05
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void setVacuumPumpOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_05,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.26: Dichiarazione del metodo `setVacuumPumpOn`

2.2.27 Il metodo `setVacuumPumpOff`

Il metodo `setVacuumPumpOff` disattiva la pompa a vuoto.

```
/** Set to OFF the vacuum pump
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 05
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void setVacuumPumpOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_05,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.27: Dichiarazione del metodo `setVacuumPumpOff`

2.2.28 Il metodo `isVacuumPumpOn`

Il metodo `isVacuumPumpOn` restituisce `true` se la pompa a vuoto è in funzione.

```
/** Is the vacuum pump ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 05
 *  @return true if the vacuum pump is ON
 *  @throw ReceiverControlEx
 */
bool isVacuumPumpOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_05
) throw (ReceiverControlEx);
```

Listato 2.28: Dichiarazione del metodo `isVacuumPumpOn`

2.2.29 Il metodo `hasVacuumPumpFault`

Il metodo `hasVacuumPumpFault` restituisce `true` se la pompa ha qualche malfunzionamento.

```
/** Has the vacuum pump a fault?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 06
 *  @return true if the vacuum pump has a fault
 *  @throw ReceiverControlEx
 */
bool hasVacuumPumpFault(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_06
) throw (ReceiverControlEx);
```

Listato 2.29: Dichiarazione del metodo `hasVacuumPumpFault`

2.2.30 Il metodo `setVacuumValveOn`

Il metodo `setVacuumValveOn` apre la valvola di pressione.

```
/** Set to ON the vacuum valve
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 07
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void setVacuumValveOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_07,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.30: Dichiarazione del metodo `setVacuumValveOn`

2.2.31 Il metodo `setVacuumValveOff`

Il metodo `setVacuumValveOff` chiude la valvola di pressione.

```
/** Set to OFF the vacuum valve
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
```

```
 *  @param port_number the port number; the default is 07
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void setVacuumValveOff(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_07,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.31: Dichiarazione del metodo `setVacuumValveOff`

2.2.32 Il metodo `isVacuumValveOn`

Il metodo `isVacuumValveOn` restituisce `true` se la valvola di pressione è aperta.

```
/** Is the vacuum valve ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 07
 *  @return true if the vacuum valve is ON
 *  @throw ReceiverControlEx
 */
bool isVacuumValveOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_07
) throw (ReceiverControlEx);
```

Listato 2.32: Dichiarazione del metodo `isVacuumValveOn`

2.2.33 Il metodo `isRemoteOn`

Il metodo `isRemoteOn` restituisce `true` se il comando remoto è abilitato.

```
/** Is the remote command enable?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 26
 *  @return true if the remote command is enable
```

```
 *  @throw ReceiverControlEx
 */
bool isRemoteOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_26
) throw (ReceiverControlEx);
```

Listato 2.33: Dichiarazione del metodo `isRemoteOn`

2.2.34 Il metodo `selectLO1`

Il metodo `selectLO1` seleziona l'oscillatore locale numero 1.

```
/** Select the first local oscillator (LO1)
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 0
 *  @param value the value to set; the default value is 0x00
 *  @throw ReceiverControlEx
 */
void selectLO1(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_00,
       const BYTE value=0x00
) throw (ReceiverControlEx);
```

Listato 2.34: Dichiarazione del metodo `selectLO1`

2.2.35 Il metodo `isLO1Selected`

Il metodo `isLO1Selected` restituisce `true` se l'oscillatore locale numero 1 è selezionato.

```
/** Is LO1 selected?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 16
 *  @return true if the first local oscillator (LO1) is selected
 *  @throw ReceiverControlEx
 */
```

```
bool isLO1Selected(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_16
) throw (ReceiverControlEx);
```

Listato 2.35: Dichiarazione del metodo `isLO1Selected`

2.2.36 Il metodo `selectLO2`

Il metodo `selectLO2` seleziona l'oscillatore locale numero 2.

```
/** Select the second local oscillator (LO2)
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 0
 *  @param value the value to set; the default value is 0x01
 *  @throw ReceiverControlEx
 */
void selectLO2(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_00,
       const BYTE value=0x01
) throw (ReceiverControlEx);
```

Listato 2.36: Dichiarazione del metodo `selectLO2`

2.2.37 Il metodo `isLO2Selected`

Il metodo `isLO2Selected` restituisce `true` se l'oscillatore locale numero 2 è selezionato.

```
/** Is LO2 selected?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 17
 *  @return true if the second local oscillator (LO2) is selected
 *  @throw ReceiverControlEx
 */
bool isLO2Selected(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
```

```
10        const BYTE port_type=MCB_PORT_TYPE_DIO,
11        const BYTE port_number=MCB_PORT_NUMBER_17
12 ) throw (ReceiverControlEx);
```

Listato 2.37: Dichiarazione del metodo `isLO2Selected`

2.2.38 Il metodo `isLO2Locked`

Il metodo `isLO2Locked` restituisce `true` se l'oscillatore locale numero 2 è *locked*.

```
1  /** Is LO2 locked?
2   *  @param data_type the type of the data; the default is 1 bit
3   *  @param port_type the port type; the default is the Digital IO
4   *  @param port_number the port number; the default is 18
5   *  @return true if the second local oscillator (LO2) is locked
6   *  @throw ReceiverControlEx
7   */
8  bool isLO2Locked(
9         const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
10        const BYTE port_type=MCB_PORT_TYPE_DIO,
11        const BYTE port_number=MCB_PORT_NUMBER_18
12 ) throw (ReceiverControlEx);
```

Listato 2.38: Dichiarazione del metodo `isLO2Locked`

2.2.39 Il metodo `setSingleDishMode`

Il metodo `setSingleDishMode` imposta la *modalità operativa* del ricevitore su *single dish* (quando un ricevitore prevede questa modalità).

```
1  /** Set the single dish mode to ON. The VLBI mode will be turn OFF
2   *  @param data_type the type of the data; the default is 1 bit
3   *  @param port_type the port type; the default is the Digital IO
4   *  @param port_number_sd the port number of the single dish mode;
5   *  the default port number is 0x13
6   *  @param port_number_vlbi the port number of the VLBI mode;
7   *  the default port number is 0x14
8   *  @param value_sd the value to turn the single dish mode ON;
9   *  default value is 0x00
10  *  @param value_vlbi the value to turn the VLBI mode OFF;
11  *  default value is 0x01
```

```
 *  @throw ReceiverControlEx
 */
void setSingleDishMode(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number_sd=MCB_PORT_NUMBER_13,
       const BYTE port_number_vlbi=MCB_PORT_NUMBER_14,
       const BYTE value_sd=0x00,
       const BYTE value_vlbi=0x01
) throw (ReceiverControlEx);
```

Listato 2.39: Dichiarazione del metodo `setSingleDishMode`

2.2.40 Il metodo `isSingleDishModeOn`

Il metodo `isSingleDishModeOn` restituisce `true` se il ricevitore si trova nella modalità operativa signle dish.

```
/** Is the single dish mode set to ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 29
 *  @return true if the single dish mode is active
 *  @throw ReceiverControlEx
 */
bool isSingleDishModeOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_29
) throw (ReceiverControlEx);
```

Listato 2.40: Dichiarazione del metodo `isSingleDishModeOn`

2.2.41 Il metodo `setVLBIMode`

Il metodo `setVLBIMode` imposta la modalità operativa del ricevitore su VLBI (quando un ricevitore prevede questa modalità).

```
/** Set the VLBI mode to ON. The SD mode will be turn OFF
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number_vlbi the of the VLBI mode;
```

```
 *  the default port number is 0x14
 *  @param port_number_sd the port number of the single dish mode;
 *  the default port number is 0x13
 *  @param value_vlbi the value to turn the VLBI mode ON; default
 *  value is 0x00
 *  @param value_sd the value to turn the single dish mode OFF;
 *  default value is 0x01
 *  @throw ReceiverControlEx
 */
void setVLBIMode(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number_vlbi=MCB_PORT_NUMBER_14,
       const BYTE port_number_sd=MCB_PORT_NUMBER_13,
       const BYTE value_vlbi=0x00,
       const BYTE value_sd=0x01
) throw (ReceiverControlEx);
```

Listato 2.41: Dichiarazione del metodo `setVLBIMode`

2.2.42 Il metodo `isVLBIModeOn`

Il metodo `isVLBIModeOn` restituisce `true` se il ricevitore si trova nella modalità operativa VLBI.

```
/** Is the VLBI mode set to ON?
 *  @param data_type the type of the data; the default is 1 bit
 *  @param port_type the port type; the default is the Digital IO
 *  @param port_number the port number; the default is 30
 *  @return true if the VLBI mode is active
 *  @throw ReceiverControlEx
 */
bool isVLBIModeOn(
       const BYTE data_type=MCB_CMD_DATA_TYPE_B01,
       const BYTE port_type=MCB_PORT_TYPE_DIO,
       const BYTE port_number=MCB_PORT_NUMBER_30
) throw (ReceiverControlEx);
```

Listato 2.42: Dichiarazione del metodo `isVLBIModeOn`

2.2.43 Il metodo `numberOfFeeds`

Il metodo `numberOfFeeds` restituisce il numero di feeds del ricevitore.

```
1 /** Return the number of feeds of the receiver */
2 unsigned short numberOfFeeds() { return m_number_of_feeds; }
```

Listato 2.43: Dichiarazione del metodo `numberOfFeeds`

2.2.44 Il metodo `openConnection`

```
1 /** Perform a TCP connection socket to the boards
2  *  @throw ReceiverControlEx
3  */
4 void openConnection(void) throw (ReceiverControlEx);
```

Listato 2.44: Dichiarazione del metodo `openConnection`

2.2.45 Il metodo `closeConnection`

```
1 /** Close the TCP connection sockets to the boards */
2 void closeConnection(void);
```

Listato 2.45: Dichiarazione del metodo `closeConnection`

2.2.46 Il metodo `isLNABoardConnectionOK`

```
1 /** Is the connection to the LNA board OK?
2  *  @return true if the connection to the LNA board is OK
3  */
4 bool isLNABoardConnectionOK();
```

Listato 2.46: Dichiarazione del metodo `isLNABoardConnectionOK`

2.2.47 Il metodo `isDewarBoardConnectionOK`

```
1 /** Is the connection to the dewar board OK?
2  *  @return true if the connection to the dewar board is OK
```

```
 */
bool isDewarBoardConnectionOK();
```

Listato 2.47: Dichiarazione del metodo `isDewarBoardConnectionOK`

2.3 Utilizzo della Libreria

La libreria `ReceiverControl` è dipendente da `MicroControllerBoard` ed è composta dalla sola classe `ReceiverControl` dichiarata e definita rispettivamente in *ReceiverControl.h* e *ReceiverControl.cpp*.

Nel listato 2.48 è illustrato un breve esempio di utilizzo della libreria.

```
#include "ReceiverControl.h"
#include <cstdlib>

using namespace IRA;

// Funzioni di conversione utilizzate solo a titolo esemplificativo:
// Converte un valore di tensione in temperatura (Kelvin)
double voltage2Kelvin(double voltage) { return voltage * 2 }
// Converte un valore di tensione in un valore di corrente (mA)
double currentConverter(double voltage) { return(10 * voltage); }
// Converte i valore di tensione nei corretti valori di VD e VG
double voltageConverter(double voltage) { return(voltage); }
// Converte il valore di tensione in un valore di pressione (mbar)
double voltage2mbar(double voltage) {
    return(pow(10, 1.5 * voltage - 12));
}

int main()
{
    // Definiamo i parametri IP e porta da passare al costruttore
    std::string dewar_IP("192.168.51.63"); // IP della scheda dewar
    unsigned int dewar_port = 5002; // Porta della scheda dewar
    std::string lna_IP("192.168.51.64"); // IP della scheda LNA
    unsigned int lna_port = 5002; // Porta della scheda LNA
    // Definiamo il numero di feed del ricevitore
    unsigned short feeds = 2;

    try {
        // Istanziamo l'oggetto ReceiverControl
        ReceiverControl rc = ReceiverControl(
```

```
31            dewar_IP,
32            dewar_port,
33            lna_IP,
34            lna_port,
35            feeds
36        );
37
38        // Avvia la calibrazione (con checksum per default)
39        rc.setCalibrationOn();
40        cout << "Il generatore della marca di rumore e' attivo? " \
41             << (rc.isCalibrationOn() == true ? "si" : "no") << endl;
42
43        // Disabilita il controllo errori (nessun checksum)
44        rc.setReliableCommOff();
45
46        // Valore della pressione prima della conversione
47        cout << "Valore in Volt: " << rc.vacuum() << endl;
48        // Valore della pression dopo la conversione
49        cout << "In mbar: " << rc.vacuum(voltage2mbar) << endl;
50
51        // Temperatura della vertex prima della conversione
52        cout << rc.vertexTemperature() << endl;
53
54        // Temperature criogeniche
55        cout << "Valore della prima temperatura criogenica: " \
56             << rc.cryoTemperature(1, voltage2Kelvin) << endl;
57        cout << "Valore della seconda temperatura criogenica: " \
58             << rc.cryoTemperature(2, voltage2Kelvin) << endl;
59
60        // Accendi gli LNA dei canali left
61        rc.turnLeftLNAsOn();
62
63        // Leggi i FetValues
64        ReceiverControl::FetValues values = \
65            rc.fetValues(0, 4, currentConverter, voltageConverter);
66
67        // Leggi gli StageValues
68        ReceiverControl::StageValues svalues = rc.stageValues(
69            ReceiverControl::DRAIN_VOLTAGE,
70            4,
71            voltageConverter
72        );
73
```

```
74          // Imposta la modalita' operativa VLBI
75          rc.setVLBIMode();
76          cout << "La modalita' operativa VLBI e' attiva? " \
77               << (rc.isVLBIModeOn() == true ? "si" : "no") << endl;
78
79          rc.closeConnection(); // Chiudi la connessione
80      }
81      catch(ReceiverControlEx& ex) {
82          cout << ex.what() << endl;
83          return 1;
84      }
85      return 0;
86  }
```

Listato 2.48: Esempio di utilizzo della libreria `ReceiverLibrary`

Bibliografia

[1] Alessandro Cattani e Andrea Maccaferri. Descrizione della scheda ALISRT (SRT LNA Bias Board). Private communication.

[2] Alessandro Cattani e Andrea Maccaferri. Reciver 22 GHz Multifeed LNAs Controls. Private communication.

[3] Alessandro Cattani e Andrea Maccaferri. Recivers Dewar Controls. Private communication.

[4] Franco Fiocchi, Marco Morsiani, Giuseppe Maccaferri, e Andrea Orlati. Protocollo di Comunicazione per Schede a Microcontrollore. Rapporto interno IRA 358/04.

Elenco dei listati

Indice analitico