Software di controllo per il Digital Base Band Converter: Software di comunicazione con la FPGA

A.Melis[1], G.Comoretto[2]

[1]INAF - Osservatorio Astronomico di Cagliari

[2]INAF - Osservatorio Astrofisico di Arcetri

Arcetri Technical Report N° 5/2010

Abstract

Il Digital Base Band Converter è una piattaforma hardware basata su logiche programmabili, sviluppata all'IRA di Noto, che è stata adottata da tutti i radiotelescopi della rete EVN come backend standard per le osservazioni VLBI.

Si compone di più schede impilate tra loro per l'acquisizione e l'elaborazione di segnali radioastronomici e da un computer con il quale è possibile configurare il sistema e interfacciarlo con il mondo esterno. Grazie alla riconfigurabilità delle FPGA di cui dispone, questa piattaforma può essere utilizzata per realizzare sistemi di acquisizione o di analisi dati differenti.

In questo documento illustriamo il software di controllo per il sistema. Oltre alle classi che gestiscono la comunicazione a basso livello, viene descritto anche un programma che funge da server generico. Il server permette il controllo di un generico applicativo tramite una serie di comandi testuali, inviati al sistema tramite una socket.

1 Introduzione

Il Digital Base Band Converter (DBBC) è una piattaforma hardware adottata, dalla comunità EVN, come standard per la gestione delle osservazioni VLBI. La descrizione del sistema si trova in [2]. Si tratta di una macchina molto versatile e riadattabile a diverse configurazioni quali polarimetri, total power, spettrometri, spettropolarimetri ([3]) ecc.
In questo documento descriviamo il software utilizzato per configurare e comandare il sistema, sia in locale che da remoto.
Daremo una descrizione del codice utilizzato partendo dalle classi più a basso livello fino ad arrivare al programma con il quale il DBBC opera in modalità server che accetta connessioni da client esterni.
Il sistema operativo installato al momento della stesura di questo documento è Windows XP ma quello definitivo sarà SUSE Linux.

2 Software di controllo

In questo capitolo andiamo a vedere le classi, sviluppate in linguaggio C/C++, che ci consentono di controllare il sistema.

2.1 Classe DBBCInterface

La classe a livello più basso è **`DBBCInterface`**, definita come segue:

```
class DbbcInterface
{
protected:
   I16 card_num;
   I16 err;
   I16 card;
```

```
public:
	DbbcInterface();
	~DbbcInterface() ;
	int Error() const { return err;}
	int Write(int pos, int core_reg, int block_mask, int value);
	U32 Wread(int pos, int core_reg, int block_mask, int value);
	U32 Read(int pos, int core_reg, int block_mask);
	int ReadN(int pos, int core_reg, int block_mask, U32 buf[], int num);
};
```

Quando eseguiamo un'operazione di lettura o scrittura, dal PC di controllo comunichiamo direttamente con la prima scheda del DBBC; questa fornisce i segnali di controllo, configurazione e programmazione per il sistema.
La scheda si interfaccia al computer di controllo utilizzando un'interfaccia parallela PCI7200 [4], e questa classe gestisce la comunicazione a basso livello con la prima scheda del DBBC.
Oltre a costruttore e distruttore, vengono definiti i metodi per la comunicazione con l'interfaccia; per inviare un comando a quest'ultima, si utilizza un valore a 32 bit che viene posto nella porta di uscita dell'interfaccia. Il significato dei bit in questo valore è descritto nel modo seguente:

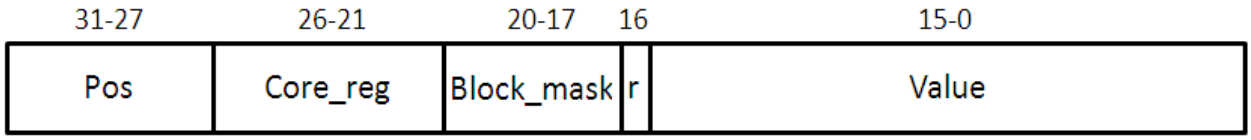

Figura 1 : Campi, e relativo range in termini di bit, del comando da inviare all'interfaccia

I campi hanno il seguente significato:
Pos: Indica la scheda (Core o Core2) sulla quale vogliamo effettuare l'operazione.
Core_reg: Indica il registro sul quale effettuiamo l'operazione.
Block_mask: Indica il numero del modulo VHDL all'interno della FPGA sul quale vogliamo effettuare l'operazione.
R: Read (1)/Write(0) bit
Value: Indica il valore da scrivere sul registro **`Core_reg`**.

Le temporizzazioni della scheda sono mostrate in fig. 2. Il segnale di Trig abilita le operazioni di lettura o scrittura, che sono selezionate dal bit 16 della porta di uscita DO. La porta DI viene utilizzata per leggere un registro interno alla FPGA, a 32 bit.

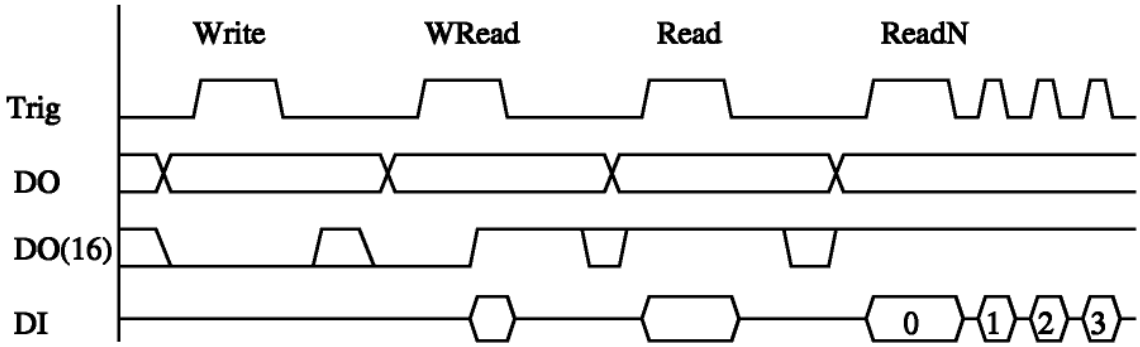

Fig. 2: temporizzazione degli accessi all'interfaccia

2.1.1 Costruttore

Dopo aver messo a zero gli slot, viene richiamata la funzione **Register_Card**, che ha la seguente firma:

```
I16 _stdcall Register_Card (U16 Card_Type, U16 Card_num)
```

I16 è un tipo **short**, **_stdcall** è usata per chiamare le funzioni API Win32, U16 è un tipo **unsigned short**, **Card_Type** è il tipo di scheda che si vuole inizializzare (PCI 7200 nel nostro caso) e **Card_num** è il numero di sequenza di slot per le schede sul bus PCI; la prima scheda inserita nello slot PCI è 0, la seconda è 1 e così via.
La funzione inizializza l'hardware della scheda, e restituisce un identificativo numerico univoco per ogni scheda installata; tale identificativo deve essere utilizzato in ogni chiamata successiva per poter accedere alla scheda che si è appena registrata. Se il valore restituito è diverso da card, viene stampato un messaggio d'errore e il programma si chiude.
La funzione **Register_Card** deve essere chiamata una volta sola all'inizio del programma utente, e deve esistere una singola istanza di questa classe nel sistema, che gestisce la comunicazione con tutti i moduli FPGA presenti.

2.1.2 Distruttore

Richiama semplicemente la funzione **Release_Card(card)**, che ha la seguente firma:

```
I16 _stdcall Release_Card (U16 Card_Number)
```

dove **Card_Number** è il numero della scheda da rilasciare.
La funzione rilascia le risorse utilizzate dalla scheda, viene usata quando l'applicazione non usa più la scheda o quando l'utente chiude il programma.

2.1.3 Error

Quando una chiamata ad una funzione produce un errore, i metodi di lettura/scrittura ritornano una condizione di errore. Il corrispondente codice di errore viene salvato nell'attributo **err**, che

può essere esaminato attraverso questo metodo.

2.1.4 Write

Il metodo serve per le operazioni di scrittura, e per ogni operazione bisogna usare comandi a 32 bit, con il formato indicato in fig. 1, inviati attraverso la porta parallela.
Il metodo assembla una parola (identificata con D0) a 32 bit, attraverso operazioni di “AND” e shift logico, a partire dai parametri **pos**, **reg**, e **mask** della chiamata. Il valore da scrivere nel registro così identificato, troncato a 16 bit, viene posto nei 16 bit meno significativi della parola.
E' importante sottolineare che il bit 16 serve a distinguere le operazioni di scrittura da quelle di lettura, quindi viene sempre posto a 0 nelle operazioni di scrittura.
Prima di scrivere nella porta di uscita dell'interfaccia, viene pilotata la linea di trigger, utilizzando la funzione **D0_WriteExtTrigLine(card,1)**; questa funzione è disponibile solo per la scheda PCI7200, la sua firma è:

I16 D0_WriteExtTrigLine(U16 Card Number, U16 Value)

e setta la linea di trigger alta (**Value=1**) o bassa (**Value=0**).
Dopo che il trigger è stato alzato, si invia il valore assemblato alle linee di uscita, attraverso la funzione:

I16 D0_WritePort(U16 Card Number, U16 Port, U32 Value)

dove **Card Number** è il numero della scheda sulla quale vogliamo scrivere, **Port** è il numero della porta d'uscita (0 nel caso della PCI7200) e **Value** è il valore da inviare sulla porta (D0 nel nostro caso). Quindi il trigger viene pulsato basso, e riportato alto.

2.1.5 Wread

Questo metodo serve per le operazioni di scrittura e rilettura, cioè utilizzando come numero di registro il medesimo: per esempio scrivo sul registro 1 e leggo dal registro 1.
Dopo aver svolto le operazioni della parte di scrittura, identiche a quelle del metodo **Write**, viene effettuata una seconda scrittura, con il bit 16 posto ad 1 per abilitare la lettura, e viene invocata la funzione DI_ReadPort(card,0,&DI), la cui firma è:

I16 DI_ReadPort (I16 Card Number, U16 Port, U32 Value)

Questa funzione legge i dati dalla porta di ingresso e li ritorna come valore di ritorno del

metodo.

2.1.6 Read

Con questo metodo andiamo a leggere un registro. La strategia utilizzata è quella di effettuare una scrittura fittizia per “puntare” un registro per poi leggere da quel registro; viene quindi invocata la scrittura su un determinato registro ma con il bit 16 posto a 1, ossia bloccando la scrittura stessa.
Dopo aver inserito **pos**, **core_reg** e **block_mask**, andiamo a shiftare D0 di 16 bit e poniamo a 1 il bit 16, in modo che si abbia D0=XXXXXYYYYYYZZZZ10000000000000000. In queste condizioni i 16 zeri non vengono scritti sul registro indicato, ma viene comunque selezionato il registro da cui vogliamo leggere, operazione fatta con la successiva chiamata a DI_ReadPort.

2.1.7 ReadN

Questo metodo serve per effettuare letture multiple, cioè per leggere più volte dallo stesso registro. Questo serve per velocizzare la lettura di grosse moli di dati (es. uno spettro), assieme ad una logica di autoindirizzamento posta nella FPGA che presenta in sequenza ad ogni successiva lettura tutti gli elementi di un array di dati.
I valori da leggere vengono memorizzati in un buffer **buf** di **num** elementi; dopo aver “puntato” il registro con la solita scrittura fittizia, si esegue un ciclo di num letture che riempie il buffer.

2.2 Classe Interface_FPGA

La classe **Interface_FPGA** eredita da **DBBCInterface**, e definisce i metodi per la scrittura e la lettura sulle FPGA delle schede Core/Core2. Si tratta essenzialmente di un “wrapper”, una classe (quasi) vuota che consente di utilizzare in modo consistente software scritto su macchine differenti. In particolare questa classe serve per adattare l'hardware del DBBC al software scritto per la scheda TFB di ALMA[1].
Allo stesso modo è possibile utilizzare una versione di questo oggetto in cui la comunicazione con la DBBCInterface avvenga tramite il server descritto nel cap. 3, rendendo indipendente il software posto a livello più alto dalla sua collocazione fisica (all'interno del sistema DBBC o in un computer esterno).
Nei metodi di questa classe vengono richiamati opportunamente i metodi di DBBCInterface (la corrispondenza dei membri è board=pos, module=block_mask, addr=core_reg, val=value).
La definizione della classe è quella che segue:

```
class Interface_FPGA :
      public DbbcInterface
```

```
{
protected:

public:
	Interface_FPGA(void);
	virtual ~Interface_FPGA(void);
	int WriteFpga(int board, int module, int addr, int val);
	int ReadFpga(int board, int module, int addr);
	int WReadFpga(int board, int module, int addr, int val);
	int ReadFpgaBlock(int board, int module, int addr, U32 buf[], int num);
};
```

Il corpo di costruttore e distruttore è vuoto, in **WriteFPGA** viene semplicemente richiamato il metodo **Write** di **DBBCInterface**, in **ReadFPGA** si richiama il metodo **Read**, in **WreadFpga** si richiama il metodo **Wread** e in **ReadFpgaBlock** si richiama il metodo **ReadN**.

2.3 Classe DBBC_component

La classe DBBC_component viene utilizzata per descrivere e controllare un component, cioè un generico strumento hardware con funzionalità arbitraria (uno spettrometro, un filtro digitale, un convertitore BBC...) implementato all'interno di una FPGA. La classe è virtuale, deve cioè essere utilizzata come classe base per ciascun componente specifico, e rappresenta solo una "cornice" uniforme alle funzionalità proprie di questi ultimi.
Un component viene identificato da un indirizzo, che specifica il numero della scheda (**board**), il numero del component VHDL (**module**), e il primo indice dei registri (**reg_init**), normalmente a zero. In questo modo si può implementare più componenti all'interno di una stessa FPGA, sia utilizzando il campo **block_mask** che assegnando a ciascun componente un sottoinsieme dei registri disponibili.
La classe implementa la comunicazione con le FPGA sfruttando l'oggetto **Interface_FPGA** ed alcune operazioni generiche.

La definizione è la seguente:

```
class DBBC_component{
protected:
   int board;
   int module;
   int reg_init;
   Interface_FPGA *intf;
public:
   DBBC_component(Interface_FPGA *a_intf, int addr, double clock_freq=128.e6);
```

```
    virtual ~DBBC_component() {}
    virtual int Init();
    int WriteFpga(int addr, int val);
    int ReadFpga(int addr);
    int ReadFpgaBlock(int addr, U32 buf[], int num);
};
```

2.3.1 Costruttore

In fase di istanziazione dell'oggetto ne viene specificato l'indirizzo hardware addr, composto dal numero della scheda (**board**), dal numero del component VHDL (**module**), e dal primo indice dei registri (**reg_init**). Viene inoltre specificata, tramite un puntatore, l'**Interface_FPGA** da usare per la comunicazione con l'hardware, ed un **double** che indica la frequenza del clock (128 MHz).
Quindi, ad ogni classe che controlla il relativo component VHDL, deve essere fornito un address dove sia specificata la scheda Core/Core2 e il numero del component istanziato nell'unica FPGA presente.
Il metodo **Init()** inizializza il componente. Essendo le operazioni da effettuare specifiche di ciascun componente, il metodo è virtuale, cioè viene definito in modo specifico nelle sottoclassi. Una funzione che deve essere implementata in tutti i componenti è il reset del PLL per la generazione del clock, alzando ed abbassando la linea specifica.

I valori di **board=b**, **module=m** e **reg_init=ii** vengono “estratti” da **addr** utilizzando il formato (esadecimale) **0xbmii**. Quindi i valori per **board** e **module** sono interi a 4 bit, e il primo indirizzo utile è a 8 bit.

2.3.2 WriteFPGA, ReadFPGA, ReadFPGABlock

Questi metodi sono relativi ai corrispondenti metodi dell'interfaccia, con la differenza che la classe gestisce automaticamente l'indirizzamento alla FPGA e al blocco di registri propri del componente. Gli unici parametri da specificare sono l'indirizzo del registro su cui scrivere (addr) e il valore da scrivere (val).

3 Software per l'utilizzo del DBBC come server

Per il controllo da remoto, si utilizza una **socket** che rimane in attesa che qualche client esterno faccia richiesta di connessione.
Per questo scopo si è realizzata una serie di classi che incapsulano una comunicazione client-server di tipo testuale o binario. Naturalmente è possibile utilizzare qualsiasi altra libreria di classi con funzionalità analoghe.

3.1 Gerarchia di classi per la gestione di una connessione client-server

Le classi che compongono questa gerarchia sono:

3.1.1 Socket Connection

```
class SocketConnection
{
  public:
  // ritorna un puntatore ad una riga di testo
  // Il metodo rientra dopo che sia stata ricevuta una riga completa, terminata da \n
   char * GetLine()
  // Trasmette una riga di testo, convertendo il \0 finale in un \n se newline=true
   const char *PutLine(const char * line, bool newline=true);
  // trasmette  e riceve un buffer binario. Length e' espressa in bytes
   int PutBuffer(const int * line, const int length);
   int GetBuffer(const int * line, const int length);
   int Flush();
   bool CloseConnection();
   bool IsConnected();
};
```

La classe gestisce le operazioni di comunicazione sulla socket.
In particolare la **GetLine** effettua un buffering dei dati ricevuti, attendendo finché non sia arrivata una riga di testo completa, terminata da “\n” o “\0”. Alloca quindi un buffer di memoria sufficiente a contenere la riga intera, e lo restituisce al chiamante. Il programma chiamante ha il compito di deallocare la memoria quando la riga di testo non sia più necessaria.
I metodi **GetBuffer** e **PutBuffer** si preoccupano di continuare la comunicazione fino a quando il numero di bytes specificati non sia stato scritto/letto completamente.
I metodi **CloseConnection** e **IsConnected** servono per chiudere la connessione e per controllarne lo stato. La connessione viene anche chiusa automaticamente dal distruttore, e operazioni di lettura/scrittura su una connessione chiusa ritornano errore.

3.1.2 Socket Server

```
class SocketServer: public SocketConnection
{
public:
    // Costruttore.
    // address è una stringa che contiene l'indirizzo IP
    // port è il numero di porta della connessione IP
       SocketServer(const char *address, int port);
```

```
   //  attende la richiesta di connessione del client
       bool WaitConnection();
};
```

La classe implementa una comunicazione di tipo server. I parametri da passare al costruttore di questo oggetto sono l'indirizzo IP del server e la porta sulla quale attendere la richiesta di connessioni.
Una volta instanziato l'oggetto, questo viene posto in attesa di una richiesta di connessione tramite il metodo **WaitConnection**. Quando un client richiede la connessione, il metodo ritorna ed è possibile usare i metodi ereditati da **SocketConnection** per comunicare con il client. La connessione è terminata dal metodo **CloseConnection** in uno dei due lati.

3.2 Programma di server generico

Questo programma serve per controllare un sistema generico basato sul software DBBC da parte di un server esterno. Il server può essere un programma specifico per una applicazione composta da component definiti, o al limite una sessione di **telnet** con cui si programmano e leggono registri individuali.

Viene innanzitutto istanziato un oggetto di tipo Interface_FPGA, per interagire con l'hardware. Dopo aver definito la porta alla quale ci si deve connettere, viene istanziato un oggetto di tipo SocketServer che si occupa di gestire la connessione.

Si attendono quindi connessioni con il metodo WaitConnection di SocketServer.
Quando arriva una richiesta si entra in un ciclo in cui si riceve una linea di comando inviata dal client. Se la linea è nulla, la connessione viene automaticamente chiusa, e il sistema torna in attesa di nuove connessioni col metodo WaitConnection. Se comincia con la lettera E, il server termina. In tutti gli altri casi, si passa il comando alla funzione “processline”, descritta più avanti.
Eseguito il corpo della funzione processline, il programma trasmette al client la risposta ricevuta, come valore esadecimale, ed attende nuovi comandi

3.2.1 Funzione processline

La funzione processline effettua il parsing del comando inviato dal client ed utilizza le funzioni di Interface_FPGA per comunicare con le schede Core/Core2.
I parametri da passare alla funzione sono il puntatore alla stringa che contiene il comando ricevuto, l'indirizzo dell'oggetto di tipo Interface_FPGA e l'indirizzo dell'oggetto di tipo SocketServer.
La riga in input viene scomposta in cinque interi che rappresentano rispettivamente il tipo del comando, la scheda sulla quale vogliamo lavorare, il numero del component su cui operiamo, l'indirizzo di lettura/scrittura e il valore **val** da scrivere in caso di scrittura.

Abbiamo 5 tipi di comando, identificati con un valore numerico da 0 a 4:

- *0*: Restituisce il valore **`val`** che viene scritto, senza comunicare con l'hardware (funzione di eco, usata per debug della comunicazione)
- *1:* Richiama il metodo **`WreadFpga`** e restituisce il valore riletto.
- *2:* Richiama il metodo **`ReadFpga`** e restituisce il valore letto.
- *3:* Vengono effettuate letture multiple con il metodo **`ReadFpga`**; i valori letti vengono memorizzati in un buffer binario che viene inviato al client. Viene restituito il numero di valori letti.
- 4: Vengono effettuate letture multiple con il metodo **`WreadFpga`**; i valori letti vengono memorizzati in un buffer binario che viene inviato al client. Viene restituito il numero di valori letti.

4 Bibliografia

[1] G. Comoretto, A. Russo: “Software di comunicazione con il correlatore Altera”, Arcetri internal report 2/2007

[2] G. Comoretto, G. Tuccari: “Reference design for the Digital BBC Architecture”, Arcetri internal report 2/2008

[3] A.Melis, G.Comoretto: Porting of a spectropolarimeter on Digital Base Band Converter, Arcetri internal report 3/2009

[4] ADLINK PCI7200 programming manual

Indice generale